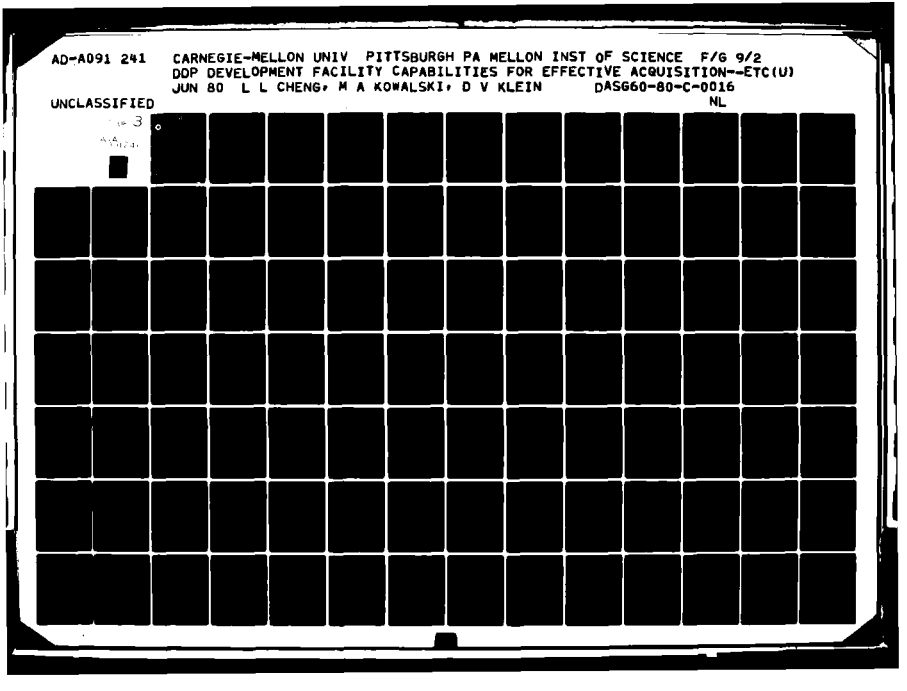
AD-A091 241 CARNEGIE-MELLON UNIV PITTSBURGH PA MELLON INST OF SCIENCE F/G 9/2
DDP DEVELOPMENT FACILITY CAPABILITIES FOR EFFECTIVE ACQUISITION--ETC(U)
JUN 80 L L CHENG, M A KOWALSKI, D V KLEIN DASG60-80-C-0016
UNCLASSIFIED NL



MELLON INSTITUTE

COMPUTER ENGINEERING CENTER

A Division of Carnegie-Mellon University

Mellon Institute
COMPUTER ENGINERING CENTER
4616 Henry Street
Pittsburgh, Pennsylvania 15213

DDP Development Facility Capabilities for Effective Acquisition of DDP Software and Hardware (Draft)

Submitted by:

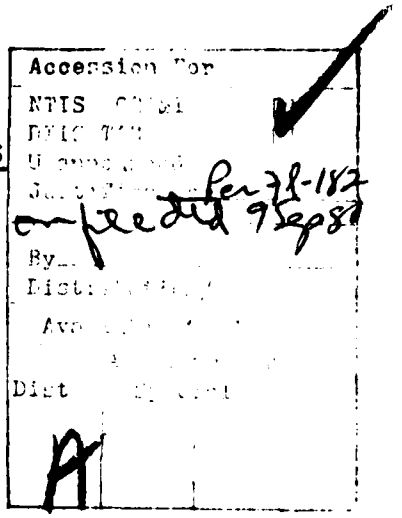
L. L. Cheng
M. A. Kowalski
D. V. Klein
M. L. Soffa
W. T. Sze
P. Vansickle

June 1980

219

Sponsored by
Ballistic Missile Defense Systems Command
Contract Number DASG60-80-C-0016

225500



CONTENTS

SECTION 1

INTRODUCTION

1. INTRODUCTION

1.1. Purpose of Study

This report documents the results to-date of the study titled Distributed Data Processing (DDP) Software/Hardware Support Plan in support of the Ballistic Missile Defense (BMD) Terminal Phase. This study was initiated mid-January 1980 and its purpose is to identify the capabilitites required by BMD in order to have high confidence that full scale development (FSD) of a DDP subsystem can be initiated and completed within a short (3 to 4 year) time frame. The study is directed toward the definition of a DDP development facility for BMD that will help achieve the goal of a speedy development.

The BMD DDP subsystem referred to in this report is assumed to be general in nature, i.e., its components may be hardware or software. Hardware components may be general purpose or specially developed for this application; they may already exist or are yet to be developed. Software components may consist of a combination of high order languages, assembly languages and microcoded modules.

1.2. Context and Scope

A distributed system is one in which multiple processors, memory and peripherals are interconnected into an architecture specially suited to a particular application. Control of processing functions and computing resources is usually distributed among a number of processors. Data may also be distributed. Computing resources in a distributed system may also be physically situated at geographically distributed locations. As is expected, the distributed nature of a system adds much complexity to its structure, and its development.

Distributed systems is a relatively new and untested field. Many questions about their design, implementation, and verification remain unanswered. Established methods and tools are few and not yet well-accepted. The potentials of a distributed system, however, are promising. By tailoring distributed components of system architecture to fit the application, higher processing speed, higher data throughput, better fault tolerance, modifiability and ease of extension can be achieved. Previous BMD efforts have therefore led to the identification of the need for a distributed data processing subsystem in the Terminal Phase of BMD layered defense system. The Advanced Data Processing Subsystem Investigations (ADPSI) was initiated to maintain a state of readiness for the eventual effective deployment of a distributed data processing subsystem.

The study described in this report is part of the ADPSI effort. To exhaustively investigate all possible capabilities suitable for a BMD DDP subsystem development is naturally beyond the scope of this study. The emphasis will therefore be placed

on tools and techniques that 1) pertain to the distributed nature of the system, and 2) help minimize the development time. It should be noted here that optimizing development time may conflict with speed which is another important goal of the BMD DDP subsystem. In the case of conflicts, a decision needs to be made based on trade-offs. Special emphasis is placed on system development and software although hardware acquisition will also be addressed.

SECTION 2

DEFINITION OF A DDP SOFTWARE DEVELOPMENT FACILITY

2. DEFINITION OF A DDP SOFTWARE DEVELOPMENT FACILITY

The BMD Data Processing Subsystem is a complex real-time system that must perform complicated operations on massive quantities of data within very severe time contraints as well as survive hostile conditions. The development of such a system is involved enough without the added constraint of a very stringent development time frame. A facility that has the capabilities to help the managers, system designers, software developers, hardware developers, and system integration/testing personnel will be of grave importance towards the realization of the timely development of a system of quality. The following are steps that should be taken to define an effective development facility.

2.1. Investigations and Analyses

A good facility in support of distributed system development is an integrated collection of tools that will ease the functions performed during the development process. It is, therefore, itself a tool. To define such a facility for a BMD subsystem, it is necessary to have a good understanding of the BMD application requirements, the distributed development process itself, and tools useful to this process.

2.1.1. BMD Environment

Since the software development facility to be defined is to support the development of a BMD subsystem, a good understanding

of the nature of the BMD application is imperative.

2.1.2. DDP Development Process

The development of a distributed system is more involved than that of a traditional centralized system because of its complicated structure and the higher degree of freedom in the choice of a solution architecture. A better understanding of the major steps involved in the distributed system development process and the issues pertinent to each step will provide better groundwork for the definition of a facility to support it.

2.1.3. DDP Development Tools

During the development of distributed systems, the use of appropriate tools will be very important; and, tools of real value will necessarily have to be sophisticated in order to support the complex activities to be performed. An investigation of tools that may be of use for DDP development will indicate more clearly the capabilities a development facility must be able to provide.

2.2. Definition of Capabilities

After the foregoing preliminary investigation, one is then in a position to define the software development facility in terms of the capabilities it must provide. Capabilities are functions performed by the facility to serve the user. For instance, access to a high order language with certain characteristics, or an emulator with certain features, are capabilities a development facility can provide the user.

2.3. Selection of Specific Items

A capability may be realized in a number of ways. The need for a high order language with certain characteristics, for example, may be satisfied by a number of existing languages. An emulation capability with certain features may be implemented in a variety of hardware and software choices. Selection from such alternative implementation approaches must be made so that the definition of a development facility will be further refined to contain a list of specific tools to be used.

2.4. Hardware Support

The tools so identified may require the support of special hardware, and such hardware needs to be explicitly stated to complete the definition of a development facility.

2.5. Scope and Direction of Current Study

The approach taken by the current study was based on the above mentioned steps. The work began with a study of the steps and activities involved in the distributed system development process, including considerations and tradeoffs typical of most distributed systems (see Section 4). Potential tool categories useful to DDP systems development were then identified. Their relevance to DDP development was examined (see Section 5).

BMD requirements have been studied more carefully in other ADPSI efforts. Hence, the current work does not attempt to further analyze BMD requirements in detail. Some assumptions were made about the BMD environment and its DDP subsystem (see Section 3). These assumptions, together with the studies of

distributed systems development and tools, were used to define the needs of development facility capabilities for BMD.

To perform an in-depth study of all tool categories identified was beyond the efforts allocated to the current study. Several tool categories that were considered most important were therefore selected for further study, namely, design languages for distributed systems, implementation languages for distributed systems and their environment, and emulation/simulation facilitites (see Sections 6, 7, 8, and 9).

Analyses of work covered in Sections 4, 5, 6, 7, 8, and 9 led to the identification of capabilities recommended for inclusion in a BMD DDP development facility (see Section 10). The recommended capabilities define a BMD DDP development facility in terms of the functions it must provide to the user as explained in Section 2.2. Alternative ways to implement such capabilities will be discussed in the relevant Sections and, where possible, specific choices will be recommended. Intelligent choices in some cases may not be possible within the scope of this present task because of the lack of information and time. After all the specific choices have been made, hardware support identified, and a plan set up for their acquisition, the development facility so defined will be ready for installation. A subsequent report will present a plan and schedule for the orderly acquisition of capabilities identified in this report.

Note that the development facility recommended as a result of this study is based on investigations into a few of the more important tools required for distributed systems development. It is by no means the result of exhaustive examination of all

capabilities desirable in a DDP development facility. The approach taken, however, did have in mind the overall picture of the totality of other necessary tools. The detailed studies were performed under the broad framework of the overall requirements of a development facility. The objective is that as more detailed investigations are undertaken in the other tool areas such as testing and management, the definition of the development facility will be easily expanded to include the other tools to achieve a more complete definition of an effective development facility.

SECTION 3

BMD ENVIRONMENT

3. BMD ENVIRONMENT

3.1. Assumptions of BMD Characteristics

The assumptions made about the BMD environment and the BMD DDP subsystem requirements are listed below. These assumptions are based on information obtained from review meetings, conversations with ADPSI personnel, and parts (pages 152 to 289) of a document (Number 17773-10751019C, title unknown) dated 1 July 1976 obtained from McDonnell Douglas Corporation.

1) The BMD DDP subsystem will be non-geographically distributed.

2) The design of the radars will be considered as outside the scope of the DDP subsystem.

3) Development of the DDP subsystem includes requirement definition, system design, implementation of software and hardware components, testing, and certification.

4) The required response time from beginning to end of the endo-atmospheric Terminal Phase is 30 seconds. This includes initiation, detection, tracking, discrimination, and interception.

5) Only digital hardware and software will be considered as components. Human intervention is

ruled out because of response time requirements, and analog equipment is beyond the scope of this study.

6) The requirements state that only a specified number of items have to survive attack.

3.2. BMD Software

There are at least three types of software that need be developed. One is the application software which is to perform the specified functions for the BMD DDP subsystem and to interface with the other parts of the Layered Defense System. Another type of software is test software which simulates parts of the system not available for testing purposes. For instance, a test driver may be necessary to simulate the threat or the other parts of the Layered Defense System that the DDP subsystem need to interface with. Or, there may be a need to simulate radar data for testing. Yet another type of software is operating system type of software. The reasons that operating system software has to be developed are:

1) software is needed to control the distributed hardware resources.

2) software is needed to schedule processing functions.

3) because super high processing speed is required, the overhead of general purpose operating systems is to be avoided in the target system.

4) cost of software is small compared to that of hardware because the hardware cost is repeated for each defending site in the order of hundreds.

SECTION 4

DDP DEVELOPMENT PROCESS

4. DDP DEVELOPMENT PROCESS

This Section reports the results of a brief analysis of the distributed systems development process. Some special characteristics of distributed systems are listed below; the major phases of the development of a distributed system and the related activities within each phase are then discussed.

4.1. Distributed System Characteristics

The following characteristics typify most distributed systems, though some characteristics are not unique to distributed systems alone.

1. Exploitation of special architectures and processing elements to fit an application is possible. This means that more than one processing element may be employed, each with the possibility of its own memory and peripherals. Or, memory and peripherals may be shared among several processing elements.

2. Processing functions may be assigned to either hardware or software.

3. Concurrent functions inherent in the application implies the need for concurrent processing concepts.

4. The sharing of data and resource among processes or processors implies the need to guarantee mutual

exclusion of access.

5. The scheduling of processes are more complex.

6. Inter-processor communication is necessary. This is usually achieved via shared memory or messages.

7. There are more degrees of freedom in hardware and software selections.

8. There are more system performance parameters of interest than a traditional centralized system, and system performance is much more complex and difficult to measure.

4.2. Requirements Phase

Several types of requirements are specified for a distributed system.

- Processing requirements specify the functions or operations that need to be performed.

- Data requirements specify the amount and nature of data to be processed and produced. The origin of certain input data may indirectly specify the location of a node.

- Performance requirements specify the system characteristics that need be measured and the levels of performance they must achieve.

- Constraint requirements specify any physical limitations such as size and weight of the system that must be satisfied, and any environmental conditions within which the

system must operate.

The above requirements must be analyzed with regard to

- completeness - is any part of the requirements missing, and are they specified to enough detail for design and implementation?
- consistency - does one part of the requirements contradict another part?
- feasibility - do the requirements specify a system that can be built?

4.3. Design Phase

The design phase is perhaps the most conceptually complex phase of any system development. The advent of structured programming and modern systems engineering practices have shown that if proper attention, time and effort are given to the design phase of software development, cost savings will be achieved in the implementation, testing and maintenance phases of the software life cycle. The activities involved in the design phase of a distributed system are much more involved than those of a conventional system. The design of a distributed system includes the design of each component which is a computer system in itself, and the network of the system which connects the software and hardware component parts into a functional whole.

The design phase of distributed systems development involves several activities.

4.3.1. Elaboration

This activity includes the elaboration of requirements into higher degrees of detail, the assignment of performance, feasibility assessment, and verification planning.

4.3.2. Partitioning

The partitioning activity logically groups processing functions by commonality [Mariani 79]. It also defines the interfaces among the logical groups. Commonality criteria vary. One example is the nature of the processing functions. Processing functions performing similar operations may be logically grouped together. Or, processing functions that are parts of the same task as seen from a higher level of system description may also be grouped together. Access of common data is another example of a commonality criterion for partitioning.

4.3.3. Allocation

The partitioning function above does not indicate how the logical groups are to be assigned to computing resources. Logical groups of processing functions that access common data, for instance, may be assigned to the same processing element for ease of data access. However, they may also be deliberately allocated to separate processing devices to maximize parallellism. The choice in large part depends on the guidelines of the overall objectives of the system.

The partitioning and allocation activities in combination assign processing requirements to computing resources [Mariani 79].

4.3.4. Tradeoff Analyses

The above activities essentially result in the definition of processing nodes. These nodes can be connected together in a variety of ways. Each topological configuration will have its own characteristics in behavior, performance, and its need for hardware support and software control. The problem can be stated in the following fashion [Boorstyn 75].

Given: nodes (for each candidate partition)
processing required
traffic
performance required
constraints

Find: topology
links - internode communication

Typical Objective Functions:

capacity thruput	(maximize)
response time	(minimize)
cost	(minimize)
(initial, development, and maintenance)	

Other Performance Considerations:
reliability/fault tolerance
modifiability
absence of deadlocks
ease of verification
expansion possibilities

maintenance

non-technical considerations

operational personnel required

Tradeoff Considerations:

Many degrees of freedom in distribution of

control	- communication, scheduling
physical data base	- speed, data thruput, coherency
memory at nodes	- mutual exclusion, speed

Assignment of capacity to various nodes to

fit architecture to processing needs

Choice of hardware capacity and processors

Means of communication among processors

1. shared memory	- mutual exclusion of access
2. messages	- protocols
	- traffic load and pattern
	- routing in case of failure
	- synchronization

Tradeoff between hardware and software

freedom of assigning processing function to HW or SW

each topological configuration implies additional

software needs such as

- memory required
- complexity (affects cost, development time, and quality)

)ISTRIBUTED SYSTEM DESIGN

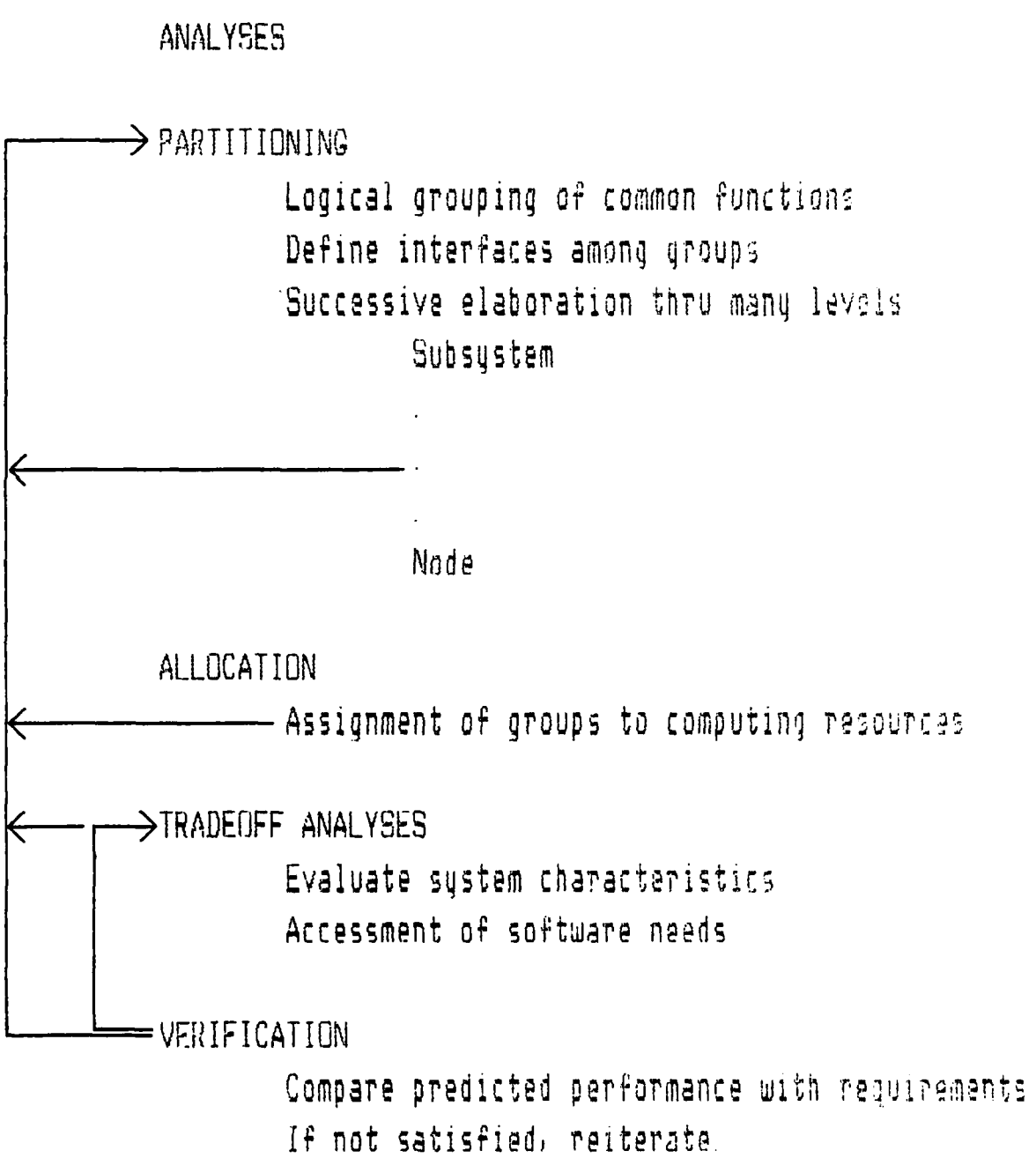


Figure 1. Distributed Systems Design.

4.3.5. Verification against requirements

After tradeoff analyses have been performed, the predicted attributes of the candidate system have to be compared against the stated specifications to see if the performance requirements will be satisfied. If more than one candidate topology satisfies the stated requirements, then they will be evaluated according to cost, development time, or other pertinent attributes for final selection. If a candidate architecture does not satisfy the stated performance, then alternate topologies will be considered, or the application will be re-partitioned to arrive at an acceptable solution. The activities of elaboration, partitioning, allocation, and tradeoff analyses are therefore iterated until a satisfactory solution emerges.

The need for iteration is also brought about by the fact that all these design activities need be performed for successive levels of design. After a solution architecture is designed to a certain level of detail, analyzed to ensure fulfillment of specifications, it is then expanded into a more detailed design by elaborating on each component part.

The distributed nature of the solution makes it possible to further elaborate each component part independently of one another. Concurrent design of various portions of the system mostly likely takes place in the develpment of a distributed system. One obvious example is that after the overall network of nodes and their interconnections are specified, further design of each node can take place concurrently. This also applies to the concurrent design of hardware and software modules as long as at the verification step the predicted performance of the entire

system is matched against specified requirements.

4.3.6. Two Design Approaches

Two design approaches to distributed design worthy of note are mentioned here. One noticeable characteristic common to both of these approaches is that many activities take place in distributed system development before hardware and software are even identified. This confirms the previous discussions on the complex nature of tradeoff considerations in distributed design. The indication then is that the design phase of distributed development is more prolonged, more important, and deserves much attention.

Total System Design Methodology

Though the Total Systems Design Methodology (TSDM) [Clark 79] concept, developed at Rome Air Development Center (RADC), was originally intended to be applied in the design of general systems, consisting of a combination of hardware, software, firmware, or manual components, it is very applicable in the realm of distributed systems.

It stresses initial functional conceptualization of a system regardless of whether these functions will eventually be implemented in hardware, software, firmware, or in a manual procedure. It also emphasizes analysis, iteration and early feedback as important means of coping with the complexity of concurrent system design. In fact, TSDM isolates the high level design activities to form a new step called Decomposition, whose function is to provide the transition from a set of functional

requirements definitions to the design of a system in terms of hardware and software components. In other words, Decomposition allocates abstract functional requirements to physical resources.

Air Force procurement procedures have traditionally separated computer system requirements at the start into software and hardware components to accomodate long leadtimes in hardware procurement, often at the sacrifice of software quality or the integrity of the entire system. TSDM was proposed to correct the error in this systems design philosophy, and advocates a "software first" concept. The benefits of postponing hardware/software implementation decisions are maximized, and the ultimate solution, as represented in the design of the system, is one that is natural to the problem it addresses.

Simulation is used heavily in TSDM, both at the requirements definition level as well as at the more detailed levels of design to ensure integrity of the whole system.

Baseline Approach

A distributed design technique was developed at TRW as the result of analysis of the distributed design process [Mariani 79]. It advocates four activities - Analysis, Partitioning, Allocation, and Synthesis - to be applied at four different levels of design, viz., the Subsystem, Nodal, Computer Systems, and Hardware/Software levels. Analysis consists of elaboration, requirements identification, performance assignment, feasibility assessment, and verification. Partitioning identifies commonalities of processing entities and groups together processing requirements. Allocation relates characteristics of

partitions to computing resources. Synthesis considers interprocess links, data base, control, topology, protocol, performance evaluation and verification.

The Partitioning and Allocation activities together allocate processing requirements to computing resources and are therefore similar to the Decomposition step in TSDM.

The baseline approach advocates that the four levels of design fit into the overall phases of distributed systems development as follows:

REQUIREMENTS PHASE

- DDP Requirements

PRELIMINARY DESIGN

- Distributed Architecture Design
  - Network Design
  - Node Design
  - Computer System Design
- Integrated Hardware-software Design

DETAILED DESIGN

INTEGRATION AND TEST

- Implementation and Test
- Integration and Test

OPERATION AND MAINTENANCE

4.4. Implementation Phase

The design phase is ended when the target system is specified in terms of hardware and software components and their interconnections; and each hardware software component is specified to sufficient detail for implementation. Acquisition of the hardware and software parts can then take place. Detailed design of each component part takes place within this phase.

Implementation of these component parts can take place concurrently, i.e., hardware acquisition and software acquisition can occur simultaneously; acquisition of different processors can also overlap, as does the acquisition of different software modules.

From the sheer number of independently identifiable subparts of a system, a distributed system offers more opportunities for concurrent acquisition than is possible in a centralized system. Unless the entire system is developed by one contractor, the coordination required by the procurement procedure will be horrendous. A systematic method of procurement should be mapped out as procurement guidelines to ease the managerial functions of the procurement. On the other hand, three arguments stand against the contracting of the entire system to one vendor. One is that it may be desirable or even necessary to acquire hardware items from a different vendor than one that can provide software items. Second is that the entire system may be too large for one vendor. Third is that the managerial function is pushed back on the shoulders of the vendor although in this case coordination across contracts is not necessary.

4.5. Integration and Testing Phase

Testing of each component part must necessarily take place during the implementation phase. These parts must then be integrated, hardware and software alike, together so that the entire system can be tested as a whole. Granted that in the development of a distributed system, not only can design and implementation of various nodes or subsystems take place concurrently, testing of individual nodes and subsystems can also be done in similar fashion. In the interest of compressing full scale development time, testing and validation should be done in a stepwise manner as is highly possible in a distributed system. Portions of the distributed system should be tested as they become available and the entire system can be validated at various levels. This will simplify the final validation process at the end of full scale development.

4.6. Verification and Certification Phase

This is the phase in which the procuring agency verifies the completed target system and certifies its acceptance. To do this, field testing, simulation and other verification tools might be used.

4.7. Overlapping Phases

In the development of a centralized system, the above discussed phases are more clear-cut. In distributed system development, if the opportunities for concurrent development are fully exploited, some of these phases may overlap. One node or subsystem may be in the design phase while another may be in the process of being implemented and tested.

SECTION 5

DDP DEVELOPMENT TOOL CATEGORIES

5. DDP DEVELOPMENT TOOL CATEGORIES

This Section reports the results of a task to identify tools and techniques that will best support the activities and attributes of a distributed system as outlined in the preceding Section. The emphasis is on tools that will help shorten the development time. Tools in this context include any methods, concepts, languages, hardware, and automated software applicable to decreasing the length of the development process. The reasons for the usefulness of each identified category will be given.

5.1. Systems Design Tools

As the previous Section has indicated, the distributed system development process is characterized by the complex design trade-offs and the numerous activities that take place before hardware/software components are identified. For this reason, any tool that will facilitate the design of a distributed system is bound to help shorten the time and effort for the total development of the system.

5.1.1. Design Techniques

Design techniques proposed to support distributed systems are few. The Baseline Approach discussed in Section 4.3.6 is one example. Other design techniques, however, that are intended for the design of systems in general or for the high-level design of software can be 'borrowed' for application in distributed system

development. An example is the Total System Design Methodology, also discussed in Section 4.3.6. Other examples are briefly listed below.

Levels of Abstraction

The concept of levels of abstraction was defined by Dijkstra [Dijkstra 68] to cope with the inherent complexity of software systems. This technique reduces the overall system logic to manageable parts and provides a conceptual framework for achieving a clear and logical design. The entire system is viewed as a hierarchy of levels, each supporting an important abstraction. Higher levels may appeal to lower levels to perform tasks; lower levels, however, are not aware of the existence of higher levels. Each level of abstraction is composed of one or more related functions that share common resources. The external functions may be referenced by higher levels while the internal ones are strictly used within the level itself.

This concept was first applied in the development of the 'THE' multiprogramming system by Dijkstra. Since then, its usage has been widespread and hardly a venture in the application of newer software engineering techniques goes by without the inclusion of levels of abstraction.

Top-down Design

Top-down design [Mills 71, Yourdon 75] refers to a method of design by top-down expansion, beginning with a simple expression of the target system describing the dependencies among major components, each of which is an abstraction of a process or resource. These components are in turn decomposed into further abstractions one level at a time until the whole system is

defined. Interfaces among these components are explicitly defined as each component is recognized. These interfaces include data connections and the services provided.

Top-down design makes use of the concept of levels of abstraction, but is not identical to it. The former stipulates a direction of arriving at the design of a system, while the latter does not as long as the resulting design can be conceived in terms of abstractions.

Reactions to top-down design are generally favorable, although it is recognized that the design pocess is ultimately iterative in nature. Elshoff, in reporting his experiences with top-down design [Elshoff 74], found some difficulty in adhering to strictly top-down design in the identification of common utility routines used throughout the system. He tended to identify them first, which would be contrary to an absolute top-down approach. Dijkstra also pointed out a similar issue in his discussions of the design of 'THE' system. He suggested that sometimes knowledge of the lower levels, such as how a machine works, does help a designer make the correct decisions at the higher levels. It would eliminate some iterations. In practice, therefore, some iteration is almost unavoidable if a good design is to be achieved.

Parnas' Modularity

In conventional modularization, the criterion used in decomposing a complex function into modules is to make each major step in the process a module. Parnas' criterion for decomposition [Parnas 72] is "information hiding" or the "black-box approach". The modules no longer correspond to steps in the

processing. Each module is characterized by its knowledge of a design decision unknown to the rest of the system. The interface of each module is chosen to reveal as little as possible about its internal implementation.

Stepwise Refinement

Program development by Stepwise Refinement is a method proposed by Wirth [Wirth 71] in which the creative activity of programming, as distinguished from coding, is considered as a sequence of design decisions. Each step is a refinement of the previous step to bring about the decomposition of tasks into subtasks and of data into data structures. Each refinement thus implies a series of design decisions; program and data specifications are refined in parallel.

Structured Design

The underlying consideration of Constantine's structured design technique [Stevens 74, Constantine 67] is to isolate more simple independent modules with minimal connections among them. The result is that these modules are not only easier to comprehend, easier to implement, less error-prone, but also less repetitive because they are more likely to be reusable without recoding, thus avoiding duplicate code.

In [Stevens 74] the authors discuss the strength of association among modules, cohesiveness, and binding. They advocate that design be performed at three levels: system, program and module. There are trade-offs involved in structured design. "The overhead involved in writing many simple modules is in the execution time and memory space used by a particular language to execute the call. The designer should realize the

adverse effect on maintenance and debugging that may result from striving just for minimum execution time and/or memory." [Stevens 74]. According to Stevens et al, depending on the actual overhead of the language being used, it is also possible that a structured design can result in less execution and/or memory overhead rather than more because of the tightness of the design, better organization in general and less duplicate code.

Data Abstraction

Data abstraction deals with the ability to represent data in abstract form regardless of the internal representation or implementation on the machine. In the same way that successive decomposition of a complex problem into abstract functions simplifies the problem, the successive refinement of the data involved also simplifies the problem to be solved. One should be able to represent data in an abstract form that is most natural to the original problem. Then at each design step the data is made more specific until finally it is defined in terms of data structures offered in the programming language of choice.

Hand in hand with the idea of abstraction is that of data encapsulation which specifies that as a datum is defined, so should the operations that are allowed on it.

Early efforts in the data abstraction area were made by Dijkstra [Dahl 72], Wirth [Wirth 71]. Liskov and Zilles [Liskov 74]. Wulf [Bentley 79]. and attendees of a conference on Data Abstraction, Definition and Structure [Proc 74].

Most of the modern programming languages such as Pascal, Modula, Ada, Concurrent Pascal, allow user-defined data types to

support data abstraction.

Programming-in-the-Large vs Programming-in-the-Small

DeRemer and Kron [DeRemer 76] advocate distinguishing the activity of global detail from that of detailed design. They proposed a Module Interconnection Language (MIL) for programming-in-the-large to express the overall program structure of a large system in a concise, precise and checkable form. A MIL may be regarded as a higher-level language capable of resolving the interconnectivity of program modules much the same way a 'linker' prepares for 'loading' a program. One consequence of this approach is the need for a moe complex compiling system. Feedback on a crude version of this compiler indicated that its benefits more than compensate its cost.

5.1.2. Mathematical Analysis Techniques

Formal mathematical disciplines such as queuing theory, statistical analysis, convex programming, projective geometry and permutation analysis have been applied to network design. These disciplines are all applicable to the network design of distributed systems to predict and evaluate proposed topologies.

5.1.3. Simulation

Simulation has been used in a wide variety of areas. In the realm of distributed systems design, simulation is a useful tool to model the real-time behavior of asynchronous processes within the system.

5.1.4. Design Languages and Analyzers

Design languages and their accompanying tools will be treated in better detail in Section 6.

5.1.5. Design Automation

Two tools, proposed by TRW [TRW 79], are aimed at automating the partitioning and allocation steps of the design process. They are called the Partitioning Model and the Allocation Model respectively, and are designed for operation on the VAX. The status, performance, and guidelines used for these tools are not clear to us at this point.

5.2. Software Development

5.2.1. Concurrent Languages

To support DDP naturally, with all its inherent concurrent processes, a language capable of expressing such concurrency will greatly enhance the ease of coding. The importance of a concurrent language will be treated in greater depth in Section 7.

5.2.2. Programming Support Environment

This category includes programming aids such as editors, debugging aids, program library management aids. They will be discussed in Section 8.

5.2.3. Software Engineering Techniques

Most of the design techniques discussed in Section 5.1.1. are applicable to detailed design and coding and will not be

repeated here.

The virtues of structured programming need not be extolled once more. Suffice it to say that much has been learned from its introduction into the software field and the effect it has had on the design, implementation, testing, verification, and maintenance of software. It also had affected the design of new programming languages.

5.2.4. Compiler-writing Tools

Since designers of a distributed system will always be on the lookout for cost-effective processors that will best fit the application at hand, the chances of selecting a relatively new processor are high. The code generation portion of existing compilers will have to be modified so that the chosen implementation language can be compiled into machine code for that processor. To this end, compiler writing tools are important. [Dunbar 75, Feltman 76, TRW 73] are examples of work on compiler writers.

The YACC (Yet Another Compiler-Compiler) project at Bell Laboratories provides a general tool for controlling the input to a computer program. The YACC user describes the structures of the input, together with code which is invoked when each structure is recognized. One version is coded in the C programming language and another version is in Ratfor. YACC is implemented in C under the UNIX operating system. It is claimed to be reasonably portable.

The original purpose of YACC was to help as a design tool for the front end of compilers, i.e. lexical scanning, parsing.

It has been applied to other problems with success. Basically YACC is used to build parsers that interact with a lexical analyzer and output phases of a program. YACC handles LALR(1) grammars with disambiguating rules.

The grammar input is in the form of productions. Associated with each production is an action. An action is implemented by a subroutine call to a C language routine with parameters from the production. Successful uses of YACC have been applied in the construction of a C compiler for the Honeywell 6000, a system for typesetting mathematical equations, low level implementation language for the PDP-11, and APL and BASIC compilers to run under UNIX. Another project worthy of review is the PQCC (Production Quality Compiler-Compiler) project at Carnegie-Mellon University, although still at the development stage. The goal of the project is to create a truly automatic compiler writing system that generates quality code. Emphasis is placed on 1). production of optimized code by applying well-understood optimization techniques; 2). friendliness to the user; and 3). a reduction in production and maintenance costs and time for language implementation and support. The goal is to produce a compiler for a given language on a given machine in three man-months.

The phases of PQCC are developed to perform some particular task of optimization on the TCOL representation of the source program. The phases will run serially and are designed to be independent of each other. TCOL is the univeral intermediate representation of the source program. An abstract tree-structured representation will be passed from phase to phase. The various transformations performed by each phase will not alter the format

of the TCOL representation but only the structure of the tree that represents the program.

The PQCC has some well defined limitations in order to keep the project within reasonable bounds of feasibility. It only supports algebraic languages such as Algol, PL/I, Fortran, Bliss, C, Pascal and Ada. It does not support languages with data structures such as strings in Snobol, processing in LISP, array processing in APL. Target machines are limited to one address (PDP-8, NOVA), two address (PDP-11), three address (VAX), and general register (PDP-10, 1103, IBM/370) machines. It does not support pure stack machines and other highly specialized architectures.

5.3. Hardware Development

The availability of hardware development tools will enhance the possiblities of developing special-purpose processors to tailor the architecture to the application. Such tools include hardware descriptive languages, well surveyed and summarized in [Shiva 79, Dudani 79], and being standardized in the CONLAN effort.

Included in this category also are logical design tools, physical design tools, and analyses tools. It is not the intention of this study to cover existing hardware development tools, but will cite one example. The SCALD system [McWilliams 78a, McWilliams 78b, McWilliams 78c]. used in the development of the Lawrence Livermore Laboratory S1 computer, consists of a computer-aided logic design subsystem, a physical design subsystem, and a timing verifier which is vital to the S1

development because of the ECL components used. Some SCALD software requires a large quantity of memory, making it impossible to operate on many machines. It has been successfully used on the Mark 1 version of the S1 processor to help design and develop Mark II.

5.4. Management Tools

At the systems level, management tools are important to distributed systems design to help manage the concurrent development of hardware and software components, and the concurrent development of individual hardware and software parts. The expected difficulties in managing the procurement of these system components as discussed in Section 4 will be alleviated to some degree with the use of good management tools.

At the software development level, good management tools have been found to be necessary and useful. One of the experiences learned from the development of the HXDP distributed system is that the software management was much more difficult than the hardware management [Jensen 80]. The fact that they had good people who are experienced in the management of large software development attributed much to their successs.

Several techniques used in software development management that have shown documented results are given here.

Chief Programmer Team

Chief programmer team is a method of organizing a programming team around a skilled and experienced programmer who performs critical parts of the development work. He specifies and integrates various parts of the system to be built. Each

part can then be coded and checked out by other members of the team. In this way, the talent of the chief programmer is utilized fully; he also has the perspective of the whole subportion of the system being developed.

Favorable results of the application of this technique is reported in [Baker 72].

Code Reading

Weinberg discusses 'egoless programming' in his book titled Psychology of Programming [Weinberg 71]. His idea is that programmers should not mind the discovery of their errors by other people. Reading one another's code is beneficial for several reasons: one has a more objective viewpoint in reading code written by another person because one usually does not make the same assumptions as the author; emotional fluctuations of one person can be normalized if the code he wrote on a bad day is checked by another person.

This technique helped uncover a fair amount of logic errors early (before the code is tested) in an experiment [Corrigan 75] to evaluate software project management techniques. Syntax errors are caught by compilers easily and were found not to worth the effort of human detection. Logic errors, on the contrary, were missed by compilers and should be the errors to look for in code reading.

Structured Walk Throughs

The idea of structured walk throughs is similar to that of code reading with the exception that it can also apply to design. Review meetings are held during which one team member will

explain his design or code. Team members will raise questions and check the design or the logic of the code. In the case of review of design, documentation of design is necessary, such as design languages.

Favorable experiences are reported in [Barrett 76].

5.5. Testing and Validation

In a distributed system, because of the parallel development efforts, testing or validation can be performed in a stepwise fashion in the interest of shortening development time. Portions of the system can be validated as they become available and the entire system is validated at various levels of detail.

Proof of correctness [London 75] is a technique worth monitoring though its practicality in large systems has yet to be demonstrated. Perhaps critical portions of programs can be verified by adding assertions to be proved.

Symbolic execution is another testing technique used on conventional systems. It operates on a set of input values in one execution. If the domain of inputs and range of outputs are properly considered, symbolic execution can give a high level of confidence that a program is correct for a large set of inputs [Howden 76, Miller 75, King 75].

Simulation is also used heavily for testing purposes (see Section 9).

5.6. Specification and Documentation

The stepwise validation technique mentioned above can be realized fully only if good specification languages exist to represent the system at all levels of design and development. Such specification will also serve as documentation aid for communication among various branches of development as well as for the management of concurrent development efforts.

5.7. Emulation

Emulation is necessary to promote the development of software modules before a processor is available. It will be treated in Section 9.

5.8. Simulation

Simulation has been used in a variety of areas. In the realm of distributed systems design, it is a useful tool in modelling the real-time behavior of asynchronous processes within the system. Simulation can also be used in testing by simulating the portions of the environment within which the system will operate, e.g. simulation of threat. This topic will also be covered in Section 9.

5.9. Summary

The tool categories discussed above are summarized in a chart showing their aplicability to the various phases of the system life cycle (see Fig. 3). The Operations and Maintenance Phase, not included in previous discussions on the distributed systems development process, is included in the chart for the sake of completeness.

Techniques & Tools / DDP Development Phase	Requirements	Design	Implemetation	Integration & Testing	Verification	Operations & Maintenance
Systems Design						
Design Techniques		X		X	X	X
Math. Anal. Techniques	X	X			X	
Design Languages	X	X	X		X	X
Design Analyzers	X	X			X	X
Design Automation		X				
Simulation						
Software Development						
Concurrent Languages			X			X
Debugging Aids						
Programming Aids						
Program Library Mgmt.			X	X		X
Software Engineering Tech.		X	X	X	X	X
Compiler-writing Tools			X			
Hardware Development						
Logical Design Tools		X	X			
Physical Design Tools		X	X			
Analyses Tools		X	X		X	
Hardware Descriptive Languages	X	X	X			X
Emulation		X	X	X		
Simulation	X	X	X	X	X	
Testing				X		X
Management	X	X	X	X	X	X
Verification & Validation	X	X	X	X	X	X
Specification & Documentation	X	X	X	X	X	X

Figure 2. DDP Tools Summary

SECTION 6

DESIGN LANGUAGE AND ANALYZERS

6. DESIGN LANGUAGE AND ANALYZERS

A design language for distributed systems is a form of representation necessary to capture the design of a distributed system as it evolves so that the current status of the design is reflected in tangible form. Such a design language can serve many purposes.

6.1. Purpose of a Design Language

6.1.1. To Represent Design

The design of a system begins with specified requirements, goes through iterations of elaboration and trade-offs, and ultimately matches functions to be performed onto computing resources. In distributed systems, as pointed out in Section 4, many activities take place before components are identified to be hardware or software. For this reason, the design of a system should begin with abstract functions to be performed, and gradually as it evolves, the design will take on a more specific form that incorporates implementation decisions of the abstract functions. For a design language to be useful throughout the design phases, its scope must encompass the abstract as well as the concrete, and representation of a system should progress from general functional concepts on a global scale through varying levels of abstraction all the way to detailed specification of each module and data structure until the whole system is well defined.

One of the most important concepts in the design of a distributed system is the concept of concurrency. One reason for implementing an application as a distributed system is that of reliability and fault tolerance. Another great impetus is that by exploiting concurrent features natural to the application, a number of processors can perform tasks concurrently to provide a higher performance in speed. Unless an application has absolutely no opportunity for any concurrent processing at any level, which is somewhat difficult to imagine, there will be a need for concurrent concepts or the full advantages of distributed systems will not be available.

The concepts necessary to support concurrency and the design of concurrent systems will now be addressed.

Concurrency at the Functional Level

One of the main propelling forces for the growing importance of concurrent processing is the increasing sophistication of computer applications, especially those in the military signal-processing, command and control real-time environment. Traditionally, concurrent processing was implemented in operating systems so that resources such as the central processing unit (CPU) and asynchronous peripheral I/O devices could be efficiently utilized, perhaps among multiple users. As computers are used to help solve a wider and more complex set of problems, many of the application areas have within them innate concepts of concurrent processes. One instance of such concurrency occurs as a result of geopgraphically distributed nodes. Processing functions at these nodes take place independently of one another.

Another instance is the inherent nature of the problem in which different processes take place independently of one another in time, but may operate on common data items, regardless of geographical distribution. Thirdly, the algorithm used in the processing may be naturally decomposable into parallel operations, especially if such decomposition is implementable on a multiple-processor architecture to improve the ultimate speed of the system. A typical example is the Fast Fourier Transform algorithm often used in signal processing (see Appendix A). Such concurrency has contributed to the recent awareness of distributed systems as a solution to many problems.

Military systems, especially real-time systems such as the BMD Layered Defense System, typically contain all three types of concurrency. For instance, airborne components, sensors such as radar, weapons systems, and ground stations need to communicate with one another. The functions each component performs may be different and can continue independently as long as synchronization and intercommunication are coordinated. The response time requirements for these systems are so stringent that a distributed design for ultimate implementation on distributed architecture may be the only solution to achieving the performance desired.

Concurrency at the Architecture Level

Another factor that heightened the importance of concurrent processing is the advance in Integrated Circuit (IC) technology. With the advent of VLSI and the proposed VHSIC (Very High Speed Integrated Circuit) technology, cost-effective distributed

architectures with components as complicated as a general-purpose computer are within reach. This means that the concurrent functions inherent in an application can be mapped onto different pieces of concurrently operating hardware. This is best achieved if the natural concurrencies are retained for as long as possible throughout the design process. Any time such concurrencies are eliminated indicates that a design decision has been made.

In summary, the need for concurrency concepts stems from abstract concepts inherent in complex problems, and means of controlling multiple devices in a distributed solution architecture.

Concurrency Implications on Design

Next, we will consider the implications of concurrent processing on systems design.

Concurrent systems deal with many concepts not present in sequential systems. First and foremost is the concept of concurrent processes. Sequential systems deal with processes that take place one at a time, one after the other. Concurrent systems have to deal with processes that take place simultaneously, each of which is triggered by a particular set of events, exists for a perhaps unpredictable length of time, and is terminated by another set of events. Secondly concurrent processing involves the concept of inter-process communication since concurrent processes may need to communicate with one another, usually in the form of messages or shared memory. Finally, to describe the behavior of and relationship among these concurrent processes, to ensure that not more than one process

accesses a common resource at the same time, and to facilitate the correct sending and receiving of messages, concurrent systems design has to take into account concepts of inter-process synchronization. This is usually achieved through concepts such as busy-waits, memory interlocks, and semaphores that may be implemented in a variety of ways, including hardware interrupts and software control.

The concepts of concurrent processes, their communication and synchronization, need be dealt with at more than one level of systems design. At the higher levels, processes represent abstract functions to be performed regardless of implementation. Inter-process communication can be thought of as the data flow among processes, and inter-process synchronization as abstract methods of controlling the scheduling of processes and shared resources. At the lower, more specific levels of design, processes represent software modules or actual processors. Inter-process communication is implemented by shared memory or messages sent and received by processes. Inter-process synchronization may reflect implementation decisions that have selected the physical means of control and synchronization.

The presence of interdependent concurrent processes makes it necessary to examine aspects of a system such as the absence of deadlocks, i.e., when one or more processes are waiting for something that will not occur. Deadlocks typically occur when processes compete for common data in such a way that the data cannot be released to any process. Processes requesting resources must also be starvation-free. Other aspects of a concurrent system that are of interest include port-to-port

response times, throughput, time-delays, reliability, fault tolerance. The identification of such performance parameters is not an easy task. The acceptance level for some of these parameters is usually specified, and a designer has to ensure that the system's performance meets the specified requirements throughout the various levels of design. In concurrent systems design, a designer also has to consider tradeoffs among performance parameters of alternate topologies. Based on these tradeoffs, he can then select the topology that best satisfies the specified functional and performance requirements for the tasks to which a system is to be applied.

Thus, concurrent systems design is much more complex than the design of sequential systems. New concepts are involved, and a larger number of system aspects need be checked out and verified against requirements.

6.1.2. To Record Design Descisions

Too often, design decisions made are implicitly embedded. Decisions are made unconsciously, and other alternatives, left unrecorded, are lost. A good design language will hopefully record some design decisions more explicitly, especially when it is used conscientiously at various stages throughout the gradual evolvement of a design.

6.1.3. For Communication Among Design Team

In distributed systems design, when many concurrent development efforts take place, a design language can serve as a communication tool among the designers by representing the current design in tangible form.

The design, formally represented, can also be a means of communication between requirement definers and system designers in the verification of design against requirements. Requirement definers can be an acquiring agency or a user of the target system, or both.

On the other side of the spectrum, design languages can also serve as a means of communication between designers and coders, the ultimate form of the design being the specification for coding of software or the specifications for the development/acquisition of hardware.

Design languages also facilitate the generation of test plans by communicating the design of the system to personnel responsible for testing the ultimate system.

6.1.4. For Better Management

Section 5 has already pointed out the importance of management tools in the development of a distributed system to integrate concurrent activities. The use of a design language will only facilitate the managerial function by making the progress and status of the design apparent at any time.

6.1.5. For Testing and Validation

Concurrent activities in the development of distributed systems allow for the stepwise testing and validation of the system as discussed in Section 5. For this to be realized effectively, there must be some way of specifying the design at any stage so that validation between steps can take place. The use of a design language will serve that end.

If the design at any stage is captured by a formal language, then testing strategies can be designed long before the system is being implemented to take into account the different possible paths and different states of a system.

6.1.6. To Support Design Philosophies

There is usually an underlying philosophy in any language, especially one that deals in abstract concepts. One hears of the 'world-view' of languages such as simulation languages, and learns how easy or difficult it is to represent some concept in a certain language [Atki80].

A design language can offer a designer the power of expression of certain concepts necessary in a certain design philosophy. Or, it can be used to enforce certain rules, built into the language, to ensure adherence to standard design guidelines. The set of constructs available in a language is significant both in the capabilities provided and in the capabilities not provided. More specifically, design languages can be used to: (1) promote certain concepts in software design, such as viewing a system in varying levels of abstraction; (2) enforce certain approaches such as designing a system in a top-down manner; and (3) ensure that the ultimate design obeys certain rules that, for example, apply to the control of modules over one another and the scope of data structures.

The purposes of such enforcement and expressive power are usually to allow for better management of the complexity of a system and to promote correctness in the design.

6.1.7. To Drive Automatic Tools

The formal specification of system design in language form provides a firm basis for the analyses of systems so specified. A parser can be built to transform the specified design into some transportable machine-readable intermediate form, such as a parsed tree whose structure has already been analyzed. Automatic tools can then use that information to further analyze, report on, simulate, or validate various aspects of the proposed system. A prime example of such a tool is a simulator that examines the real-time events of the system.

6.2. Selection Criteria For a Design Language

The design of a language is strongly driven by the purpose for which it is intended, by whom it will be used, how it is to be used, and with what other tools it will be used. By defining these goals, one sets the context of the language and the scope of its operating environment. Each one of these relationships and their effect on the choice of a distributed design language were examined and will be discussed in subsequent subsections. It should be noted here, however, that these contributing factors are inter-related. The type of information a simulator is expected to provide may require that the language be able to express additional concepts otherwise not demanded by the description of a concurrent system. How a language is to be used in a design methodology will in turn affect the type of automated tools to be built and what abstractions a language has to deal with. Furthermore, the prerequisite features of a language may make it easy for a tool to extract certain information and therefore may affect the design and usefulness of a simulator because of the easy access of such information.

6.2.1. Language and Abstract Concepts

The primary function of a language is in its ability to express abstract concepts and to model real entities. A language is usually used as a communication tool, whether it be man-to-man, man-to-machine, or machine-to-machine. A language, in its purest sense, consists of a set of constructs or features, with a specified syntactic structure and semantic meaning. It is to be distinguished from software programs that might use it, e.g. simulators, compilers, interpreters. A language is also to be

distinguished from its implementation. The same programming language, for instance, can be implemented in more than one way. It can be compiled into machine code, or it can be executed by interpretation. The same language construct can also be translated into different sets of the target code.

A design language for distributed systems must be able to describe a concurrent system. It must be able to express the concepts of concurrent processing discussed in Section 6.1.1, and any additional features that may prove necessary as our understanding of concurrent processing grows. It must be able to express components of a concurrent system, its topological structure, its time-dependencies, its data dependencies and so on.

It may also need to express concepts necessitated by its other design goals since contributing factors are inter-related as was mentioned above.

6.2.2. Language and Automated Tool Support

A language, as a representation of abstract concepts and a model of real components, is passive, while an automated tool, being a software module that uses the information in the language, is an active quantity.

If a design language is to be used in conjunction with a simulator, as is the case for the proposed study, a good understanding of what information the simulator needs to provide is very critical input to the design of the language. Conversely, the simulator can also make the best of information easily accessible from the features of the design language. The

requirement for the design language under consideration is that it be able to drive a critical-event and time-sliced simulation.

6.2.3. Language and Design Philosophy

As modern software engineering techniques such as structured programming emerged on the horizon some years ago, so did a proliferation of new languages or extensions to old ones to ease the application of such techniques. The reason for this is that many of the new techniques embody new concepts or approaches to software development, and it was discovered that the old notations no longer suffice. At times they were actually hindrances to the application of good techniques. New notations were therefore required to reflect the new concepts involved. In the case of structured programming, some features, normally a standard part of a programming language, were deliberately barred from a language to prevent a user from bad practices. This is a prime example of how design and development techniques affect decisions on which constructs are to be included in or excluded from a language.

What part of the concurrent design process the design language is to support, what design philosophy it is to support, what special techniques are to be used, which level of design is being addressed, is it to drive a simulator directly or is it to be compiled into an intermediate form, are all questions that need be answered to scope the goals of a design language.

A design language can be used to support the Partitioning and Allocation tasks in the distributed systems development process as discussed in Section 4. The design language, in

conjunction with a simulator, will be able to provide early feedback to the designer as to the integrity of the proposed system and how well it meets the specified requirements.

6.2.4. Language and the User

The importance of the user need not be emphasized. As in the design of any system, the design of a language should consider user interface from the very beginning so that the resultant product will have a high level of performance and a high probability of acceptance. Ease of use, ease of learning, clarity, value as a communication and documentation tool, conduciveness to good design practices, overall quality including extendability and modifiability are all important.

In the light of the discussion above, a design language should be chosen or developed with clearly defined goals relating to the process, automatic aids, design methodologies, and users it supports.

6.3. Requirements of a Distributed Design Language

The characteristics that a design language must possess to support the design of a BMD DDP subsystem are identified. They are classified according to the selection criteria discussed above. An added characteristic that will be discussed is Experience and Status, which will give some feel for how ready and how effective a language is. These characteristics also formed the basis for comparison among design languages surveyed.

6.3.1. Concepts

Concepts refer to the abstract ideas representable in the design language, i.e., the language's ability to handle information.

Static Structure

The static structure describes that composition of a system in terms of its constituent parts and how they relate to one another.

1) Composition: a design language for BMD must be able to describe abstract functions at a high level to the specification of software and hardware components in detail.

2) Functional dependency: an abstract function must be able to be expressed in terms of other functions it calls upon to do its job. At the high level of design, this concept is necessary to support the elaboration activity of design. At

the lower level of software design, this is equivalent to the calling dependencies among modules. This is also a way of relating the functions at one level to those at a higher level, and even perhaps to the requirements.

3) Data dependencies: Data dependencies reflect the interfaces between functional components and the data usage of a component. They refer to the information passed from one part of the system to another, or shared by several parts of the system. The data that are shared may eventually be implemented as a shared variable in a program, a passed parameter, a message, or a record written and then read from secondary storage. The dependency must be describable without specifying the actual implementation mechanism.

4) Resource Ownership: a resource may be shared among processes or modules, but it may also be exclusively owned by any one processes or module. This idea of resource ownership is necessary to support design philosophies such as Parnas' information-hiding or black-box technique (see Section 5.1.1). The idea is that information should be hidden from modules that have no need to know and therefore no need to access it. Restricting the access of resources or specifying ownership (or control) explicitly in the design will prevent its unwitting or deliberate misuse.

At the detailed design level, the concept of resource ownership is commonly manifested as internal variables within a block, or as a block nested within another block.

Data

Data describes the information passing through the system.

1) Abstraction: Data must be specifiable as an abstraction at the higher levels of design to support data abstraction techniques as explained in Section 5.1.1. At one level of design, it may suffice to know that tracking information is passed from entity A to entity B, without knowing what this information consists of. At a later time, it may be necessary to say that tracking information consists of an identifier , speed and position of the tracked object. While this is an elaborated version of the above, with more details on the logical structure of the data, no implementation method has been specified as yet. Still later, it may be advisable to say that speed will be in miles per hour and it will be stored as a real number occupying x number of bytes of storage.

Data abstraction can be accomplished in one of three ways. The first is by allowing the incomplete definition of a data item, such as specifying a name for the data with details to be filled in later. Most programming languages do

not allow this. They will assume a default type of implementation such as integers if no type is explicitly specified, or they will flag it as an error. The second method is by allowing abstract types such as sets in a language to specify the logical grouping of data but not the physical implementation. The third way is by allowing user-defined data types -- a user can define the structure of a data type in terms of standard system-supplied types, and then define instances of data of the newly-defined type. Most modern programming languages offer this capability.

2) Volume and Pattern: The volume of data and the arrival pattern must be able to be specified to give a picture of the traffic in a distributive system.

Dynamic Behavior

The dynamic behavior of a system describes how a system acts as it executes in a real environment in response to data, conditions and other stimuli. It deals with the conditions under which modules are activated, the events that lead to such activation, the order in which events occur, and how these events are related to the data passing through the system. The order and timing of events are critical in a real-time system. There may be a need to indicate that certain activities take place concurrently, or that a task has to be interrupted by another task of higher priority.

1) Concurrent Processes: a design language for distributed systems has to address the concepts of concurrent processes as discussed in Section 6.1.1. Multiple instances of the same process must be allowed to act on different data at the same time.

2) Initiation/Termination Conditions and Timing: synchronization of processes.

3) Control of Common Resources: such as mutual exclusion.

4) Communication Among Processes: via messages or shared memory.

Auxiliary Design Information:

There are other information about the design of a system that will be very useful to record.

1) Verification: if a design language allows assertions at appropriate places for correctness proofs, can specify performance information, and has the ability to match system parts to requirements, then the design of the system can be readily verified.

2) Iterative Design: if a language can record possible alternative solutions, the trade-off considerations, and the rationale for the alternative chosen, it would be more conducive to iterative design. The other alternative solutions

can be reconsidered without any loss of information.

3) Implementation Information: the BMD design language must also be able to describe specific information so that the design will be detailed enough for implementation. Examples of such information are algorithms, and input and output formats.

6.3.2. Design Philosophy

This refers to the design philosophies that the design language supports.

1) World-view: some languages, for instance, are event-oriented; others are process-oriented. some languages place emphasis on tracing the data through a system. A design language for BMD DDP sybsystem must have a wide world-view to encompass all the complex concepts in the BMD Layered Defense System.

2) Levels of Abstraction: it must support the technique of levels of abstraction in the design of a system.

3) Methodology/Design Rules: it should be able to accomodate a general and flexible design approach.

6.3.3. User Interface

Format

The format of the design language must be easy to use ,also to be machine-readable.

Size and Syntax:

Ease of Understanding: the language must be easy to use and understand. The features should be intuitively understandable.

Documentation:

6.3.4. Tools

This characteristic deals with the automated tools that are available or planned to support the design lanaguage.

Data Base

It is desirable that the information represented in the design language be parsed and stored in an intermediate form so that various tools can make use of this already analyzed information. This implies that the language should be a formal one, with its syntax in Baukus-Naur Form (BNF), and its semantics well-specified. The formality of a language makes it more conducive to machine processing.

Analyzers

One type of tools that should be available is analyzers or checkers or parsers that can detect discrepancies, redundancies and incompleteness of a design.

Reporting packages

Reports should be generated to help the management of a

development effort. The management aspects of a distributed system have already been indicated as an important one within the goal of an optimal development time.

Simulators

Software that will provide statistics and gather information about the proposed system from the model as represented in the design language are usually referred to as simulators. Such tools will be discussed in more detail in Section 9.

Evaluation Tools

Other computerized tools may be built to evaluate the design of a system. Such tools may perform sophisticated mathematical analyses on the system. They may perform proofs of correctness on critical portions of the system.

6.3.5. Experience and Status

This characteristic indicates the readiness of a language for use. It includes the types of application for which a language has been used, the size of the application, and the experiences found in using it. It also includes the status of the language, i.e., whether is has been used or is still in an experimental stage.

6.4. Languages Surveyed

Several languages were surveyed for potential use as a design language for the DDP subsystem under study. Some of these languages were not all originally developed as a design language, but were felt to be suitable as a design language because of the functions they provide. They were selected for study because of their ability to represent concepts of concurrency.

6.4.1. PSL

Problem Statement Language (PSL) was developed by the ISDOS (Information System Design and Optimization System) project at the University of Michigan [Teichroew 74].

It was originally designed to formalize the requirements for a large computer-based information system, but its capabilities make it useful in high-level design as well. It has the ability to describe the static structure of a target system in terms of objects within the system, the relationships among them, including data connections. For example, PSL delineates the boundaries of the system by identifying physical units of data or information external to the proposed system. These are called Inputs and Outputs. It also identifies Real World Entities that the system interfaces with. Next, the units of data are identified, called Sets of Entities or Groups of Elements. They, together with Inputs an Outputs, represent the information flow through the system. Finally, there are Processes and their constituent processes which operate on the data.

The objects listed above are tied together by relationships such as Subpart of, Contained In, Uses, Derives, and Updates.

The ability to represent the dynamic behavior of a system is limited to describing Events that Trigger certain Processes. In addition, sizing information, some project management information, and narrative descriptions can be expressed.

Concomitant with the development of PSL was the development of a software package PSA (Problem Statement Analyzer) that builds a data base from a set of PSL statements, checks it for consistency and completeness, and retrieves as well as manipulates selected information from the data base, generating reports to the user for analysis. Reports generated include varied representations of the infomation in the data base such as selected listings, retrieval by keyword, matrices, and flow diagrams.

Previous evaluations of the language found it difficult to specify hardware components in PSL since PSL addresses functional requirements only and prevents "over-specifying". It was difficult to relate the static view of the system to the dynamic view. In other words, processes, events, and data are not related enough to give a comprehensive view of the subtleties of the dynamic system. Response times and traffic volume can only be described in textual form, not amenable to computerized parsing. Flow of information among processes is restricted because a data item cannot be both output from and input to processes. PSA is heavily reporting oriented, but PSL should be supported by more tools such as simulation to reap the full benefits of a formal specification language. The data base built from PSL statements retains much of the information for future tool support. Characteristics of PSL are listed below.

CHARACTERISTICS OF PSL

I. CONCEPTS

STATIC STRUCTURE

Composition

processes

Functional Dependency

processes can be SUBPARTS OF or

UTILIZES other processes.

Data Dependencies

processes RECEIVES input and GENERATES output;

processes UPDATES, USES, and DERIVES data.

Resource Ownership

all variables are global, hence not ownership

of data;

process ownership is indicated by SUBPARTS OF.

DATA

Abstraction

SETS consisting of ENTITIES consisting of GROUPS

consisting of ELEMENTS;

no user-defined data types or encapsulation.

Volume & Pattern

in textual form.

DYNAMIC BEHAVIOR

Concurrent Processes

no concept of different instances of the same process

working on different data at the same time.

Initiation/Termination Conditions & Timing

processes TRIGGERED by EVENTS or the presence of

INPUTS;

no timing information.

Control of Common Resources

none.

Communication Among Proceses

via data which can be messages or shared memory.

AUXILIARY DESIGN INFORMATION

Verification Information

none except performance information in textual form.

Iterative Design Information

no record of alternative solutions.

Implementation Information

no hardware specification, physical implementation of data, or algrithms.

II. DESIGN PHILOSOPHY

WORLD VIEW

process-oriented and emphasis on data flow.

LEVELS OF ABSTRACTION

yes.

METHODOLOGY/DESIGN RULES:

hierarchical; distinguishes information outside the system (INTERFACES) from that which is inside.

II. USER INTERFACE

FORMAT

textual statement; graphical reports available.

SIZE & SYNTAX

about 70 keywords or statements; no formal syntax.

EASE OF UNDERSTANDING

the world-view requires some learning.

DOCUMENTATION

textual explanations allowed.

III. TOOLS

DATA BASE

information in PSL statements are transformed into a data base for subsequent processing.

ANALYZERS

analyzes completeness, consistency, and redundancy.

REPORTING PACKAGES

good reporting package - over 15 reports including dictionary lists, DATA/PROCESS interactions, and hierachical organization of data.

SIMULATORS

none.

IV. EXPERIENCE & STATUS

PSL has been installed on a variety of machines; the language and its support tools are still being modified or extended.

6.4.2. IORL

IORL, Input/Output Requirements Language, was developed at Teledyne Brown Engineering, to reduce the costs and risks associated with the development and maintenance of computer-based systems [Teledyne 77]. It is in graphical form, with the syntax specified in Backus-Naur Form production rules. The semantics are given in graphical form.

IORL consists of three main elements, SBD (Schematic Block Diagram), IOPT (Input.Output Parameter Table) and the IORTD (Input/Output Relationships and Timing Diagram). These three elements correspond to the main steps in the progression of a systems concept through various stages of design to complete specification.

SBD's are used to identify the structure of a system in terms of components or functions and their interfaces. Interfaces can be internal to the system, as well as external, and are labelled for subsequent correlation with IOPT's.

Each IOPT expands the definition of one interface in the SBD's. An IOPT is a table that lists all the parameters in the interface by name, units, range of values and accuracy. This is just a logical definition, no physical implementation is implied.

An IORTD is a graphical elaboration of an SBD block. It uses precisely defined symbols and mathematical expressions to describe Input/Output events, their timing and precedent conditions, and processing steps between events.

User input and reporting capabilities are available in a

demonstration version. Configuration management has been designed to maintain versions and keep track of the progress of a system. Simulators are also being planned.

Characteristics of IORL are tabulated below.

CHARACTERISTICS OF IORL

I. CONCEPTS

STATIC STRUCTURE

Composition

(SBD) blocks; (IORTD) processing steps.

Functional Dependency

an SBD block expanded into IORTD steps;
an SBD block expanded into more SBD blocks
at a lower level of detail.

Data Dependencies

interfaces among SBD blocks defined by
IOPT (parameter tables).

Resource Ownership

blocks have hierarchical tree structure;
one I/O parameter table belongs to only
one interface;
internal parameter tables (IPT).

DATA

Abstraction

1 level of grouping parameters into group
numbers.

Volume & Pattern

no.

DYNAMIC BEHAVIOR

Concurrent Processes

the "And" symbol specifies concurrent processing
steps in IORTD. A set of Predefined Macros each
representing multiple I/O events can be instan-

tiated. User can define common sequence of IORTD symbols in a Predefined Process Definition (PPD) to be used over again.

Initiation/Termination Conditions & Timing

events and logical conditions control the initiation and termination of processes. Timing specified between chronologically consecutive events.

Control of Common Resources

no.

Communication Among Proceses

interfaces defined by parameter tables.

AUXILIARY DESIGN INFORMATION

Verification Information

timing specifies response time specifications; it is possible to generate event histories. Value ranges specified for parameters.

Iterative Design Information

alternatives could be indicated by "Or" symbols.

Implementation Information

Physical Interface Characteristics (PIC) planned but not complete.

II. DESIGN PHILOSOPHY

WORLD VIEW

process and event oriented.

LEVELS OF ABSTRACTION

set of SBDs, IOPTs and IORTDs specify entire system at each level.

METHODOLOGY/DESIGN RULES:

hierarchical; designed for requirements, design, implementation of general systems, and for documentation of test results.

II. USER INTERFACE

FORMAT

user input format is graphical and tabular.

SIZE & SYNTAX

syntax in BNF(Backus-Naur Form) containing 14 graphical symbols and 138 production rules.

EASE OF UNDERSTANDING

semantics requires learning, but perhaps no more so than other languages representing complex concepts.

DOCUMENTATION

IORL system documentation procedure exists.

III. TOOLS

DATA BASE

prototype system manipulates input symbols for reporting.

ANALYZERS

to be developed.

REPORTING PACKAGES

prototype s [illegible].

configuration management tools planned.

SIMULATORS

some simulator needs being analyzed;

to be developed.

IV. EXPERIENCE & STATUS

prototype system stage.

6.4.3. COSY

Lauer et al [Lauer 79] developed a new notation for specifying systems of concurrent proceses sharing distributed systems of resources. This notation, named COSY for Concurrent System, is a formal language whose syntax is given in production rules. The meaning of COSY programs was based on a translation rule which when applied to a COSY program results in a finite labelled transition net. Important properties of the programs themselves can also be studied from a collection of occurrence graphs.

Main elements of the COSY language are paths and processes. A path defines a resource and determines a collection of totally ordered set of operation activations. Concatenation and alternation are allowed in this sequence of operations on a path. Operations belonging to a component path may be active only one at a time in the order specified. In the absence of paths, the processes are permitted to activate their component operations or to progress concurrently. A macro notation, less formal and more akin to programming languages, is available to enable the conceptual decomposition of a complex system into simpler subsystems. Programs can be built using the macro notation to guarantee mutual exclusion of resources shared by concurrent processes.

The world-view of COSY is a resource oriented one in that the responsibility for ensuring proper functioning of the system is associated with the resources rather than the processes using the resources. In the process oriented approach, usually some synchronization operations are needed in the processes to ensure

correct behavior of the system. This resource oriented view is felt to simplify the task of understanding a system.

One noted deficiency is that it is impossible to specify structural criteria of absence of deadlock for COSY programs. A suggested extension is to use macro generators to replicate a number of paths and processes from given ones, thus allowing the definition of new operations in terms of already defined ones.

It is not known what types of tools are available, if any, or planned to support COSY at this time.

CHARACTERISTICS OF COSY

I. CONCEPTS

STATIC STRUCTURE

Composition

paths of operations of processes and resources; the Macro Notation allows programs scoped by BEGINs and ENDs.

Functional Dependency

Macro Notation allows conceptual decomposition of system into subsystems.

Data Dependencies

via parameters.

Resource Ownership

data defined in macro Begin ... End scope.

DATA

Abstraction

in macro notation generalization to n frames of paths by replicators.

Volume & Pattern

no.

DYNAMIC BEHAVIOR

Concurrent Processes

operations are permitted to be active concurrently unless restricted by paths.

Initiation/Termination Conditions & Timing

ordered by paths; no specific timing.

Control of Common Resources

critical sections can be built in program

to guarantee mutual exclusion.

Communication Among Proceses

only via operations.

AUXILIARY DESIGN INFORMATION

Verification Information

order of operations represented in formal basic notation which lends itself to formal verification.

Iterative Design Information

alternatives could be indicated by "Or".

Implementation Information

a suggested extended notation allows a specification section followed by an implementation section.

II. DESIGN PHILOSOPHY

WORLD VIEW

resource oriented -- resources have the responsibility to ensure that their operations are used correctly by processes.

LEVELS OF ABSTRACTION

yes.

METHODOLOGY/DESIGN RULES:

II. USER INTERFACE

FORMAT

formal statements with procedural-like macros.

SIZE & SYNTAX

macro notation contains 8 production rules in BNF.

EASE OF UNDERSTANDING

world-view requires learning, but most concepts apply uniformly to processes and resources alike.

DOCUMENTATION

language does not provided much documentaion besides information in language statements.

III. TOOLS

DATA BASE

not known at this time.

ANALYZERS

unknown.

REPORTING PACKAGES

unknown.

SIMULATORS

unknown.

IV. EXPERIENCE & STATUS

applied to illustrate many basic resource management problems. Other experience unknown.

6.4.4. Simula

Though Simula is primarily a simulation language, it has many features that warrant consideration of it as a design language. Simula will be described under simulation in Section 9.

CHARACTERISTICS OF SIMULA

I. CONCEPTS

STATIC STRUCTURE

Composition

algorithmic language which includes Algol-60. Has most of the modular structures such as blocks, procedures, functions, and classes.

Functional Dependency

module calls, can be recursive. Can pass procedures, functions and class variables as parameter.

Data Dependencies

data obey block-structure rules, permitting access to global and non-local variables. the CLASS construct can be used as a data handler which means that the class only defines data and the class can be passed from one module to another or used by remote accessing.

Resource Ownership

some restriction of access can be accomplished by textual structure of program. DEC-10 implementation has the concept of a hidden attribute which restricts access to the module in which the attribute is defined.

DATA

Abstraction

a form of data abstraction is permitted by

the class structure and the hidden facility. Also has a "virtual" facility that allows various versions of an attribute.

Volume & Pattern

DYNAMIC BEHAVIOR

Concurrent Processes

parallel processing represented logically but executed in actuality sequentially. Permits more than one instance of a module.

Initiation/Termination Conditions & Timing

Clock is a simulation clock which implicitly starts the execution of processes at the scheduled time. No real-time clock feature. No need for synchronization because of simulation.

Control of Common Resources

control is only logicaly concurrent. No need for mutual exclusion because of sequential simulation.

Communication Among Proceses

shared memory - based on static scoping and class; messages - by parameters.

AUXILIARY DESIGN INFORMATION

Verification Information

Iterative Design Information

alternatives can be flagged by the HIDDEN facility in the DEC-10 implementation of Simula.

Implementation Information

since Algol-60 is a subset of the language Simula, implementation issues can be specified with Algol.

II. DESIGN PHILOSOPHY

WORLD VIEW

process view.

LEVELS OF ABSTRACTION

classes/procedures/functions can be defined within one another.

METHODOLOGY/DESIGN RULES:

hierarchical - based on static nesting and procedure call discipline.

non hierarchical - behavior of classes when used as coroutine.

II. USER INTERFACE

FORMAT

textual.

SIZE & SYNTAX

requires 24K on DEC-10 at run-time.

EASE OF UNDERSTANDING

fairly easy to use and understand for and experienced programmer;

has good structures.

DOCUMENTATION

no explicit documentation capability.

III. TOOLS

DATA BASE

Existence of actual data base of parsed

information unknown.

ANALYZERS

based on Algol, therefore BNF grammer;

good syntax and type checking, amenable

to machine processing.

REPORTING PACKAGES

good statistical information gathering.

SIMULATORS

runs on DEC-10 and IBM 360/370.

Debugging includes break points, modifying

value of variable, and execution trace display.

Simulators tend to be quite large.

IV. EXPERIENCE & STATUS

used predominantly on simulation to model

operating system kernal and to implement

data bases.

Established compiler; ready to use.

6.4.5. Survey Summary

The characteristics of the languages surveyed are summarized in Figure 3.

The design languages surveyed were chosen because of their ability to represent concurrent concepts. Even then, they do not satisfy all the desired characteristics for describing concurrency.

Though the number of languages that can describe concurrency is quite small, the surveyed languages represent a varied sample of design languages. IORL, for instance, is graphical, while the others are textual. Simula contains a programming language as a subset. COSY has a formal basic notation, and is quite small. It is resource-oriented, while the others have a process-oriented philosophy.

Of the four languages surveyed, only Simula is an established language and ready for use. It is the view of the authors that the use of a design language will facilitate the development of distributed systems for the reasons discussed at the beginning of the Section. Since not too many suitable languages are ready for use at this time, a design language and its support tools are long-lead items whose acquisition should preceed that of a full-scale development. Since Simula is the only established language, if a DEC-10 or DEC-20 is compatible with the other requirements of a software development facility, it can easily be incorporated to serve as a compromised solution while a more suitable language can be brought up to ready state. The processing overhead associated with Simula, as with most

general purpose simulation languages, is not critical because the speed required of a development facility is not as critical as that of the target system. It, on the other hand, eliminates the development cost of a specialized simulation, and offers the flexibility for tradeoff studies that specialized simulations do not.

Ada can also be considered for use as a design language by seeing how well or poorly it satisfies the characteristics set forth here. If suitable, a design language that can also serve as an implementation language will have the added benefit of easing the transition from the design phase to the programming phase.

Another language that probably warrants investigation is RSL, a requirements language which is part of the SREM methodology developed at TRW.

FIGURE 3. DESIGN LANGUAGE SUMMARY

	PSL	IORL	COSY	SIMULA
CONCEPTS				
STATIC STRUCTURE				
Composition	process	block	process program	block procedur function classes
Func. Dep.	Yes	Yes	Yes	Yes
Data Dep.	Yes	Yes	parameters	Yes
DATA				
Abstraction	Partial	1 level	Partial	Partial
Vol. & Pattern	textual	No	No	No
DYNAMIC BEHAVIOR				
Conc. Proc.	no multiple instances	Yes	Yes	Yes
Init./Term.	Partial	Yes	order,	simulated
Res. Control	No	No	user-built	No
Proc. Comm.	Yes	Yes	via operations	Yes
AUX. INFO.	text	Partial	Partial	Partial
DESIGN PHILOSOPHY				
WORLD VIEW	process oriented	process oriented	resource oriented	Partial oriented
ABSTRACT. LEVELS	Yes	Yes	Yes	Yes
METHODOLOGY	hierarchical	hierarchical		hierarch
USER INTERFACE				
FORMAT	textual	graphical &	formal	textual
SIZE & SYNTAX	70 keywords	BNF	small	large
EASE OF UNDERSTAND.	fair	fair	fair	fair
DOCUMENTATION	textual	documentation	No	No
TOOLS				
DATA BASE	Yes	prototype	unknown	unknown
ANALYZERS	Yes	planned	unknown	Yes
REPORTING TOOLS	Yes	prototype	unknown	Yes
SIMULATORS	No	planned	unknown	Yes
EXPERIENCE & STATUS	fair	prototype	unknown	establishe

SECTION 7

IMPLEMENTATION LANGUAGE

7. IMPLEMENTATION LANGUAGE

A key consideration in the definition of a software development facility is the language in which the software is to be developed and the support tools for entering, modifying, translating, linking, loading, and testing of programs. The user and the programmer can communicate better if the communication is in terms of the job to be accomplished rather than in terms of the tools necessary to accomplish the job. Therefore, an implementation language should provide primitives which make it easy to define the objects which are typical of the application. The language should also provide control structures which make it easy to express the natural flow of control required by the application. Additionally, the detection of run-time errors and dynamic debugging in terms of the source code can be facilitated by a good language.

Good program development support tools provide an effective method of shortening the software development time and, therefore, reducing the cost. Traditional support tools are compilers, general purpose editors, linking loaders, and good testing and debugging facilities. Work being done at Carnegie Mellon University and other places is directed at improving these tools by defining a uniform and integrated system for program development. Ideally, the tools in the environment would be developed in such a way that they appear to understand each other's objectives and collaborate toward a common goal. For

instance, a syntax directed editor rather than a general purpose editor can be useful in simplifying both the processes of entering syntactically correct programs and debugging programs at the source rather than machine level.

Tools are being developed for "programming in the large." These programs will simplify the mechanics of monitoring and controlling the interdependencies of the many modules used in a large software development effort. The problem of managing a large software development project will also be eased by several projects currently underway.

This Section will cover work done concerning the selection of an implementation language for BMD DDP subsystems. Section 8 will cover work done in defining the need for a good program development environment.

7.1. Need for Concurrent Concepts

The possibility of performing computations and operations in parallel exists at many levels in a system as complex as a BMD application. At the lowest level, horizontal micro-architectures allow many micro-operations per micro-instruction. At the operating system level, Input/Output processing overlap and multiprocessing systems present the opportunity for parallel processing. At the highest level, the implementation of a BMD system as a DDP system can present even more opportunities for parallel processing.

The work being done on the 2-AU experiment at MDAC, the automatic generation of microcode, addresses the problem of detecting and exploiting parallelism at the lowest level. Parallelism at this level can be used to reduce the time needed to perform calculations. This work will no doubt be significant in shortening the computation time of various critical functions needed in the BMD application. The Very High Speed Integrated Circuits (VHSIC) program currently being funded by the DOD also addresses the problem of shortening computation time. It is expected that the VHSIC program will have a significant impact upon parallelism and computing at the micro-instruction level.

The work done at Mellon Institute has been directed toward specifying and exploiting parallelism at the operating and functional levels of a BMD DDP system rather than at the micro-instruction level. The primary goal of the work being reported was to identify a language or set of languages and the necessary hardware and software support tools needed for a system development facility suitable for the development of a typical

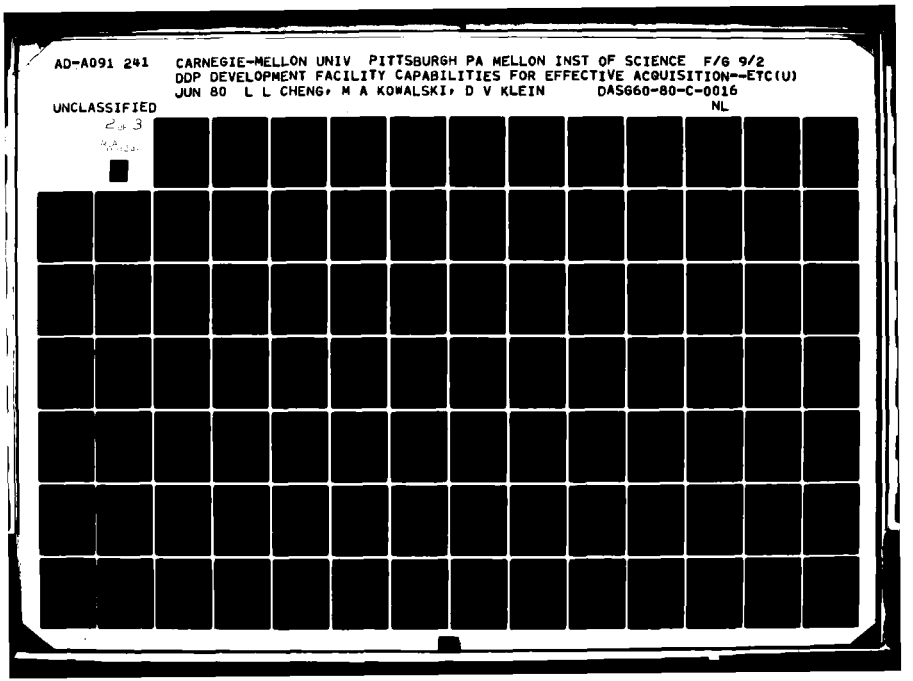
BMD application where the system is to be implemented as a DDP system.

As discussed in section 3, a typical BMD application can be classified as a real-time control system. Our approach to identifying desirable characteristics of a language suitable for a BMD DDP application has included a study of the evolution of languages developed for other real-time control applications. In particular, the portion of an operating system which must control and manage the resources (i.e. the CPU, memory, peripheral devices, and data) of the system is a real-time system. Included as Appendix B is a discussion of the use of high order languages for programming operating systems. One of the more notable requirements in the design of operating systems is the need to support concurrent processing. As many of the potential advantages of a BMD DDP system assume support for concurrent processing, we contend that an investigation into the development of languages used in writing operating systems would be beneficial.

7.1.1. Concurrency Within Operating Systems

A computer system is usually made up of many components. In addition to one or more central processing units and the main memory, it may support any number of standard I/O and storage devices as well as numerious special purpose devices such as the radar devices of a typical BMD system.

These devices are capable of operating in parallel, but at vastly different rates of speed. The time to process a single piece of data may vary from a tenth of a second for a console


AD-A091 241
CARNEGIE-MELLON UNIV PITTSBURGH PA MELLON INST OF SCIENCE F/G 9/2
DDP DEVELOPMENT FACILITY CAPABILITIES FOR EFFECTIVE ACQUISITION--ETC(U)
JUN 80 L L CHENG, M A KOWALSKI, D V KLEIN DAS660-80-C-0016
UNCLASSIFIED
NL
2 of 3

a millionth of a second for a CPU. The high differential in speed coupled with the advantages to be gained by enabling several sequential tasks to use the resources of the system simultaneously have made systems which can suport concurrent processing very desirable.

Concurrent processing at the system level is the procedure of allowing more than one logical or sequential task to be executing at a given time. This can be accomplished by sharing the system's resources such as the CPU, memory, peripherals, and data. The algorithms used to define and implement sharing of the CPU can range from simple round robin schemes to very complex procedures. Supporting the illusion that processes are running concurrently usually involves context swapping and can usurp much of the systems resources. This becomes a particularly acute problem when a system is heavily loaded.

Two factors in the development of hardware will probably have an effect upon where the illusion of tasking is supported. First, the availability of architectures which can be micro-coded will make it possible to define much of the necessary support for tasking as a hardware function, rather than requiring that such support be provided by the operating system. Second, with the availability of powerful inexpensive processing devices and the improvements in communication technology among these devices, it is becoming cost effective to support logical concurrency with actual physical concurrency such as can be provided by DDP systems.

From the point of view of the language designer however, the way in which the concurrent processing is implemented should not

be a critical factor in the specification of the language primitives. A programming language should provide primitives which facilitate the description of the function to be performed in terms of the problem to be solved. If a task force of cooperating processes is an ideal way to describe the functions to be performed, then the language should provide a convenient means of identifying and describing a task force of cooperating processes.

At a minimum, a good concurrent language should provide primitives necessary to identify functional tasks which can operate concurrently and primitives to support "safe" inter-task communication and synchronization. The compiler for the language would then translate that description into code that can execute on the the available hardware, be it either a single CPU where the illusion of concurrency is supported by time slicing, or on a DDP system where tasks actually can execute concurrently.

The possibility of exploiting the full parallelism of a DDP system presents problems which still require a great deal of research. However, it is clear that the parallelism must be addressed in the design of the system, the programming of applications for the system, and the run-time support.

7.1.2. Alternatives to Concurrent Languages

The concept of concurrent processing, as well as many language constructs, has its beginnings in operating system programming. The necessity of utilizing the various peripherals and handling the synchronization and data transfer functions between the CPU and the components in the system involves many of the same considerations that must be addressed in a DDP system.

Often applications which require concurrent processing have made use of the command language of the system in order to tap the power of the system. For example, the UNIX operating system [Ritchie and Thompson 74] provides the fork and join primitives to support concurrent processing. System calls such as LOCK, WAIT, and SLEEP can be found in the code of many data base management systems. These system calls can be used to supply the protection and synchronization necessary in a distributed data base system.

The approach of supporting concurrency by tapping the resources of a operating system has been used successfully in many applications. This approach is usually achieved by supplementing a language with a library of system macros or subroutines. There is an on-going effort at Systems Development Corporation to implement a library of network subroutines for distributed systems on a network of VAX's under the operating system VMS.

However, concurrency which is supported by a language rather than the operating system is preferable for the reasons enumerated below.

1) Concurrency supplied by an operating system implies that the application must run not only on a particular machine but under a particular operating system. Therefore, application programs which utilize operating system supplied functions are both machine dependent and operating system dependent.

2) An operating system is just another program and therefore it, too, utilizes the systems resources. In many cases, real-time control programs require only a small percentage of the functions supplied by an operating system. Therefore, it may be too costly (in terms of the resources required by the operating system) to use an operating system to support the concurrency required by a real-time control system.

3) Testing facilities for programs which contain operating system functions must simulate or emulate not only the machine but the operating system running on that machine.

4) Changes or problems with the operating system can affect the application program.

5) Interprocess communication and synchronization protection, which is defined by the language, can be checked by the compiler at compile time. This can significantly reduce the necessary run-time checking which in turn should improve the speed of execution.

6) Perhaps the most important advantage of language supplied concurrency is that functional concurrency can

be specified regardless of its actual implemetation. That would certainly reduce the cost of modifying the system over a period of time.

7) The power of a macro or subroutine is limited. There are concepts not easily implemented by a subroutine that are more easily done b, a compiler.

7.1.3. Run-time Support for Concurrency

As discussed in the previous section, mutual exclusion and concurrency are desirable. However, the actual implementation of the constructs usually require some run-time support. One approach which has been used is to support the illusion of concurrency with a Kernel, an assembly language program which runs underneath the concurrent programs. While this may be satisfactory in many cases it probably would not be acceptable in a DDP BMD application because of the slowdown in the execution speed. However, if a very efficient Kernel could be implemented in microcode or actual hardware, this might provide an acceptable solution.

The problem of shortening the system development time of a distributed system can be attacked from many angles. Up to this point we have limited our discussion to the need for a concurrent language and support tools for the design, implementation, and testing of concurrent programs. However, the target hardware can be designed in such a way that it too can be a significant factor in shortening the software development time.

The purpose of a compiler is to translate programs expressed

in one form, source code, into an equivalent program represented in another form usually the machine code of the target machine. As advances in technology make it possible to create more advanced machines, it becomes necessary to produce new compilers for the new machines. A retargetable compiler, a compiler which can produce code for different machines, is one approach to the problem. A standarized architecture or instruction set for the target machine is another approach.

Microcodable machines provide the opportunity of addressing the software development problem from "bottom-up". Traditionally, the instruction set of a machine was effectively limited by hardware costs. However, the availability of microcodable machines have expanded the possible complexity of the instruction set, so that the problem of defining an instruction set becomes that of choosing an optimum set of generally useful instructions.

This problem was addressed in the Nebula project at Carnegie Mellon University [Szewerenko and Dietz 80]. The work is part of the Military Computer Family Project of the U.S. Army initiated to design a standard computer architecture (AN/UYK-41 and AN/UYK-49). The result of the work is of particular interest for the Nebula machine is essentially equivalent to the Kernel needed to provide run-time support for concurrent programs. Nebula provides support for linked lists, execption handling, and task control. The following is a sample of the capabilities defined by a Nebula machine.

Linked list capabilities:

- insert and remove from a doubly linked list
- insert and remove from a singly linked list

Exception Handling Instructions:

- raise an exception
- store the exception code
- set the exception handler entry address
- return and propagate the exception

Task control instructions:

- loading and storing tasks
- starting and stopping tasks
- starting and stopping task with exceptions raised
- building and initiating new t contexts

Standardization of the Nebula Instruction Set will simplify both the run time support required by a concurrent language and the code generation phase of compiler design and implementation, thus, simplifying and shortening system development time.

7.2. Effectiveness of a Concurrent Language

So far the effectiveness of a concurrent language over a sequential language has been indicated primarily by experiences from the operating systems builder community. The quantitative measure of the effectiveness of a concurrent language is no easy matter. An experiment is proposed here to obtain such a measure, if one is desired, fully realizing that software experiments are typically expensive, assumptions are crude, and control of factors that might bias the experiment is difficult.

We assume that the factors affecting software quality are independent, i.e., one has no effect on the others:

1) the programmer(s);

2) the programming language;

3) the programming environment (operating systems, support tools, ... , etc);

4) the algorithm to be implemented;

5) software engineering standards (structured rules,.....,etc).

We further assume that representatives of each category can be rated in relative merit. For instance, p = P(2) / P(1) is a measure of the quality of programmer P(2) over programmer P(1).

The measures of interest are the correctness of the resulting programs and the time required to produce them, with emphasis on the latter.

The following qualities can be measured in each case.

1) x: time to code (in terms of programmer hours)

2) y: time to debug (care should be taken to keep the environment stable. One method is the use of the same language and same compiler but ruling out the concurrent features to effect a sequential language.)

3) z: time to prove (either a proof of correctness which may be impractical at present; or time for another programmer to read , understand the code, and be satisfied it is correct - subjective measure.)

Let

P(1) = programmer 1,
P(2) = programmer 2,
A(1) = algorithm 1,
A(2) = algorithm 2,
L(1) = a sequential language,
L(2) = a concurrent language,
p = P(2) / P(1),
a = A(2) / A(1),
and l = L(2) / L(1).

Holding other factors constant, two programmers of comparable ability will code two algorithms of nearly equal complexity in two languages in the following fashion to avoid "learning" of the algorithms. That is, P(1) will code A(1) in L(1), then A(2) in

L(2). P(2) will code A(2) in L81), then A(1) in L(2).

(conc. lang.)	L(2)	A(2)	A(1)
(seq. lang.)	L(1)	A(1)	A(2)
		P(1)	P(2)

Program	Algorithm	Language	Time to code	Time to debug	Time to prove
1	1	1	x(1)	y(1)	z(1)
1	2	2	x(2)	y(2)	z(2)
2	1	2	x(3)	y(3)	z(3)
2	2	1	x(4)	y(4)	z(4)

where x(1), y(1), z(1) are functions of A(1), P(1), and L(1);

x(2), y(2), z(2) are functions of A(2), P(1), and L(2);

and so on.

Then,

$$x(2) = a\ l\ x(1)$$

$$x(3) = p\ l\ x(1)$$

$$x(4) = a\ p\ x(1)$$

giving,

$$l = \sqrt{(x(2)x(3)) \ / \ (x(1)x(4))}$$

similarly for y and z, thus giving three experimental results of l, a measure of the merit (or demerit) of a concurrent language over that of a sequential language.

Note that this same experiment can be conducted to evaluate the alternative means of providing concurrency as discussed in Section 7.1.2.

7.3. Requirements of BMD Implementation Language

Section 3 indicated that critical components of a BMD system are the numeric computations and the real-time control functions. Work done by JRS industries provides further confirmation that real-time control is indeed a critical component of a BMD system.

The attributes that will affect the choice of an implementation language for BMD DDP subsystem are discussed below. The requirements for a suitable language are listed under each attribute. The list of attributes also formed the framework under which potential languages surveyed were compared (see Section 7.3).

1) Concurrent Language Features - for real-time control. The ideal language must have the concepts of processes, process initiation and synchronization, and mutual exclusion of shared resources.

2) Modularity - the BMD DDP implementation language must provide the ability to compile modules separately yet check types across boundaries. That is, each module must be able to be compiled separately, and then linked together with other compiled modules to form an executable program. This feature is desirable to promote modular development and checking, and to accommodate the need for interfacing with modules coded in other languages such as assembly language for efficiency.

3) Environment - this includes information on the machines which support the language, and support tools such as editors available to provide a friendly programming and executing environment. A complete set of good integrated program development support tools should be available for any language used to implement a system as large and complex as a BMD DDP subsystem.

4) Experience in Usage - problems in a newly defined language tend to become apparent as that language is used in various applications. There are distinct advantages in using a "mature" language which has withstood the test of time and use.

5) Readiness for Use - this includes information on available compilers and on-going or planned compiler development. A suitable language must be in a status ready for production use with good compiler support.

6) Conduciveness to Good Software Engineering Practices - a language that will provide a good program development environment to shorten development time must be conducive to the design and engineering techniques discussed in previous sections. The features that generally contribute to such support are the control structures, data abstraction or encapsulation, good error handling and recovery, and well-defined interfaces.

7) Inadequacies - this refers to the extensions that are necessary to enhance its suitability for BMD applications and the size f effort such extensions represent.

8) Performance - such as speed and reliability. The BMD language must be efficient in control of hardware resources, access of memory, context switching and arithmetic operations.

7.4. Languages Surveyed

Unfortunately, languages which do support all the features required by a BMD application do not seem to exist at this time. Thus, the choice of any language must be a compromise.

A concurrent program is a set of cooperating sequential tasks which operate simultaneously to accomplish a common objective. As previously indicated in this report, there are many advantages of designing and programming a DDP BMD system as a concurrent program. Therefore, we have investigated only languages which can support concurrent programming. The following is a discussion of the more promissing candidates:

- Concurrent Pascal (CPascal)
- Modula
- Concurrent SP/k (CSP/k)
- Ada

The concurrent features of these languages are summarized in Figure 4.

LANG.	PROCESS	MUTUAL EXCLUSION	SYNCHRON.
			DATA FCTS
MODULA	PROCESS	INTERFACE MODULE	SIGNALS - WAIT SEND AWAIT
CPASCAL	PROCESS	MONITOR	QUEUES - DELAY CONTINUE WAIT (NO. OF SEC.)
ADA	TASK	ACCEPT	SIGNALS OR PREDEFINED TASKS - SEND - WAIT - DELAY
CSP/k	PROCEDURE (CONCURRENT)	MONITOR ENTRY	CONDITIONS - WAIT SIGNAL BUSY EMPTY

Figure 4. Concurrent Language Features.

7.4.1. Concurrent Pascal

Concurrent Pascal (CPascal) [Brinch Hansen 77] represents one of the early efforts to define a language for writing "high quality" concurrent programs to implement reliable computer operating system and real-time control programs. Often, the testing of a concurrent program is very difficult because the result of a concurrent program is highly dependent on the relative speeds of its individual tasks. Unfortunately, the execution speed of a program will vary somewhat from one run to the next. Therefore, a program containing concurrent tasks may operate correctly on one run, but not on another. Locating and correcting these execution-time dependent errors are notoriously difficult.

Brinch Hansen attempted to define a programming language for writing reliable operating systems and real-time control programs in such a way that they were not dependent upon the execution speed of the tasks. The language provides for the explicit definition of any intertask communication and synchronization in such a way that it is not dependent upon the speed at which it is executed. The rules of CPascal are deliberately very strict and can be checked by a compiler. Thus, CPascal encourages or even forces reliable programming.

General Language Features

CPascal, a modification of sequential Pascal [Wirth 71], is a block structured language providing data types and operations or statements which apply to the data types. The following is a list of sequential programming facilities provided by the CPascal

language.

Declarations:

- Constant Declaration - associates an identifier with a constant.

- Type Declaration - defines the data types in terms of basic, implicitly defined, or previously declared types.

- Basic types are:
  - integer,
  - Boolean, and
  - char.

  Standard procedures to convert values of one basic type to another are:
  - ord(x) returns the ordinal value of the character x
  - chr(x) returns the character with the ordinal value x
  - conv(x) returns the real corresponding to the integer x
  - trunc(x) returns integer corresponding to the real x

- Enumeration - lists identifers that denote values constituting a data type.

- Array Structure - consists of a number of components which are all of the same component type. Each component is identified by a number of indices.

- Record - structure consisting of a number of components, called record fields. The final field of a Pascal record can be of type variant, that is, it can have values of different types.

- Variable Declaration - associates a unique name with and reserves memory for a fixed data type.

- Expression - specifies the rules for evaluating the value of

the operands being operated upon by the operators in the expression.

Statements:

- Assignment Statement - assigns the value of an expression to a variable.
- Procedure Call - denotes the execution of the specified procedure.
- Statement Sequence - is a sequence of statements separated by semicolons.
- If Statement - specifies conditional execution of actions depending on the value of Boolean expressions.
- Case Statement - specifies the selective execution of a statement depending on the value of an expression.
- While Statement - specifies the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated before the first and after each execution of the statement sequence.
- Repeat Statement - specifies the repeated execution of a statement sequence depending on the value of a Boolean expression, but the expression is evaluated after each execution of the statement sequence.
- Loop statement - specifies the repeated execution of statement sequences. The repetition can be terminated depending of the value of a Boolean expression called the exit expression.
- With Statement - specifies a statement sequence to be executed with a record variable.
- Procedure - declaration consists of a procedure heading and a block containing the declarations and statements of the

procedure.

- Function - is a procedure which returns a value.

Concurrent Language Features

Concurrent Pascal extends the block structured sequential Pascal language with concurrent PROCESSes and MONITORs. PROCESSes can only communicate by means of MONITORs. A MONITOR can DELAY a PROCESS to make their interactions independent of their speeds. Once a PROCESS or MONITOR has been correctly designed and verified, no other part of the program can make that component behave erratically as access to private and shared variables is controlled. The primitives to support concurrency are system components of the following types:

- Process type defines a data structure and a sequential statement that can operate on it.
- Monitor type defines a data structure and the operations that can be performed on it by concurrent processes. These operations can synchronize processes and exchange data among them.
- Class type defines a data structure and the operations that can be performed on it by a single process or monitor. These operations provide a controlled access to the data.
- Init statement, executed as a nameless routine, defines the access rights and allocates space for the variables of the system component being initialized.
- Routine entries provide the means of communication among system types.
- Queues may be used within a monitor type to delay and resume execution of a calling process within a routine

entry. The following standard functions applies to queues:

- empty(x) returns a Boolean value defining if a queue is empty
- delay(x) delays in queue x the calling process
- continue(x) resumes any process waiting in queue x

Program Development Environment

In addition to defining the language Concurrent Pascal, Brinch Hansen defined and implemeted an operating system SOLO written in Concurrent Pascal. Solo is a single user system intended for program development. It supports editing, compilation, and storage of Sequential and Concurrent Pascal programs. These programs can access either console, cards, printer, tape, or disk at several levels (character by character, page by page, file by file, or by direct device access).

As Solo was developed as a vehicle to illustrate the CPascal language, the editor and file system, as might be expected, is fairly primitive. The language compiler, however, is a rather sophisticated program.

The seven pass compiler for CPascal is:

- written in sequential Pascal
- generates a portable intermediate language (P-Code)
- could be modified to produce machine code
- developed for Digital Equipment's PDP-11/45 but should run with only minor modifications on any system which supports Pascal.

Run-Time Environment

In order to support the abstract concepts of monitors and processes and to interpret P-code, the output of the CPascal compiler, a compiled CPascal program, must be executed with a 4-K assembly language program called the Kernel. The kernel as designed by Brinch Hansen:

- creates and manipulates the process states
- handles process scheduling and switching
- delays a process
- continues a waiting process
- ends a process
- creates monitors
- provides mutual exclusion for monitors
- handles the entering and leaving of a monitor
- stops a job
- sets a clock
- waits a second
- indicates a system error
- does I/O

TRW has developed compilers for a modified version of CPascal to produce code for the target machine, rather than P-Code for a pseudo machine. Currently, they have compilers which can produce code for the Digital Equipment Products, the LSI-11 and the PDP-11. They are also working on a compiler which will produce code for the VAX computers. Should it be decided that CPascal would be a good implementation language, the TRW project would be of significant importance.

The existence of inexpensive microcodable machines is also a

significant factor in the run-time environment of CPascal and other high level languages, especially those which support concurrent processes. The microcodable machine can be programmed to perform many of the functions which are provided by the Kernel. Hardware support of functions that have been supplied by a software kernel should provide a significant improvement in the execution speed of CPascal programs.

Experience in Usage

- TRW has a version of CPascal for writing device drivers [Heimbigner 78]. As mentioned above, TRW has also modified the National Bureau of Standards sequential Pascal to compile CPascal into LSI-11 code, PDP-11 code, and is working on a version to produce code for the VAX. The interpreted version has been used for research in DDP systems on four PDP-11's and at the ARC.

- SUNY at Buffalo used a modified version of CPascal to implement MICROS [Wittie 79] a distributed operating system for MICRONET a network of distributed LSI-11's. Modification to CPascal are currently being implemented at SUNY in Buffalo to support dynamic tasking and dynamic memory management. SUNY at Stony Brook has also modified CPascal to support dynamic resource management [Silberschatz 77].

- The University of Berlin developed a Multi-User System In Concurrent Pascal (MUSIC) [Graef 79] with their version of CPascal.

- The University of Manchester U.K. has implemented a version

of CPascal on a CTL Modular One machine [Powell 79]. They plan to use it for a large multimicroprocessor system. Provisions for separate compilation have been proposed at the University of Manchester.

- Lund Institute of Technology of Lund, Sweden has implemented a system which used the RT-11 linker to link CPascal programs with code produced by the compiler for sequential Pascal from Oregon Minicomputer Software Inc. (OMSI) [Mattsson 80].

- Work supported by the Science Research Council (UK) at Newcastle University has been done to modify the interpreter to improve the fault-tolerant characteristics of CPascal [Shrivastava 79].

Readiness For Use

Although the compiler and kernel can be purchased from the University of Colorado, Enertec Inc. in Landsdale, PA, and probably others listed above, CPascal was not specifically designed to be used in production systems of the magnitude of a DDP BMD application. It might be a reasonable choice as an interim language while a more suitable one is being developed.

7.4.2. Modula

Modula, a language for modular multiprogramming developed by [Wirth 77], has features to support concurrent processing, encapsulation, and well-defined intertask communication and synchronization primitives. Like CPascal, Modula aims to provide a reliable high level language for operating system and process control type programming. A primary consideration of Wirth was to define a language which could be implemented efficiently.

General Language Features

Most of the sequential programming facilities of Modula have been adopted from Pascal, notably the concepts of data types and structures. Every Modula program is made of two components: the data definitions (declarations) and the algorithmic actions to be performed on the data (statements). Modula is a block structured language, where a block is a textual unit usually consisting of declarations and statements. Objects to be declared are constants, data types, data structures, variables, procedures, modules, and processes. Procedures and modules themselves consist of blocks. Hence blocks are defined recursively and can be nested. The language does not include any pointers.

As noted in the section on CPascal, a good sequential language is the basis of a good concurrent programming language. The following is a list of Modula facilities for sequential programming.

Declarations:

- Constant Declaration - associates an identifer with a constant.

- Type Declaration - defines the data types in terms of basic, implicitly defined, or previously declared types.
- Basic types are:
  - integer,
  - Boolean,
  - char, and
  - bits, defined to be an array [word length -1] of Boolean.
- Enumeration - lists identifiers that denote values constituting a data type.
- Array Structure - consists of a number of components which are all of the same component type. Each component is identified by a number of indices.
- Record - structure consisting of a number of components, called record fields.
- Variable Declaration - associates a unique name with and reserves memory for a fixed data type.
- Expression - specifies the rules for evaluating the value of the operands being operated upon by the operators in the expression.

Statements:

- Assignment Statement - assigns the value of an expression to a variable.
- Procedure Call - denotes the execution of the specified procedure.
- Statement Sequence - is a sequence of statements separated by semicolons.
- If Statement - specifies conditional execution of actions depending on the value of Boolean expressions.

- Case Statement - specifies the selective execution of a statement depending on the value of an expression.
- While Statement - specifies the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated before the first and after each execution of the statement sequence.
- Repeat Statement - specifies the repeated execution of a statement sequence depending on the value of a Boolean expression, but the expression is evaluated after each execution of the statement sequence.
- Loop statement - specifies the repeated execution of statement sequences. The repetition can be terminated depending of the value of a Boolean expression called the exit expression.
- With Statement - specifies a statement sequence to be executed with a record variable.
- Procedure - declaration consists of a procedure heading and a block containing the declarations and statements of the procedure. Standard procedures are:
  - inc(x,n) = x:=x+n
  - dec(x,n) = x:=x-n
  - inc(x) = x:=x+1
  - dec(x) = x:=x-1
  - halt = terminates the entire program
- Function - is a procedure which returns a value. Standard functions are:
  - off(b1,b2) = b1 and b2 = [] (b1,b2 of type bits)
  - off(b) = b = []
  - among (i,b) = b[i] (b is a bit expression)

- low (a) = low index bound of array a
- high (a) = high index bound of array a
- adr (v) = address of variable v
- size (v) = size of variable v
- integer (x) = ordinal of x in the set of values Defined by type of x
- char (x) = character with ordinal x

- Module - defines the primary unit of encapsulation for a Modula program. The use-list specifies imported variables and the define-list those which are exported.

Concurrent Language Features

Modula handles I/O and other machine dependent details such as "device processes (or drivers)" where as CPascal relies on the kernel, a 4k run-time support package, to handle the machine dependent details. TRW has developed a version of CPascal similar to Modula in which the user writes the device drivers. This Section defines the machine dependent and other facilities provided by Modula to express the concurrent execution of several program parts. The process, interface module, and the signals provide the basic features for concurrent execution, mutual exclusion, and synchronization.

- Process - describes a sequential algorithm intended to be executed concurrently with other processes. No assumption is made about the speed of execution of processes, except that this speed is greater than zero. Processes cannot be nested or be local to procedures.
- Interface module - is the facility which provides mutual

exclusion of access to common objects. Although similar to the monitor of CPascal, the interface module allows more than one process to be in a critical section, provided that all but one are either waiting for a signal or sending a signal.

- Signal - is similar to the monitor queues of a CPascal program. Wait, send, and test are the only operations that can be applied to a signal.
- Device module - an interface module that interfaces one or more device drivers (i.e. processes) with 'regular' processes. They, and only they, may contain a statement denoted by the identifer doio which enables I/O. While executing this statement, the process relinquishes exclusive access to the module's variables.
- Device register - serves to introduce the interface register's need to communicate with peripheral devices.

Environment

We have no information on either the program development or the run-time environments of Modula.

Performance

We have no information on the performance of a Modula program.

Experience in Usage

Modula has been used by Ford Aerospace and Communications Corporation for experiments with their Kernel Secure Operating System [McCauley and Drongowski 79].

In 1977, CORADCOM used Modula in the design and

implementation of a representative system. The project concluded with all of the representative system designed using only Modula. Modula turned out to be an important asset during the system design phase. The system also served to illustrate a potential for a significant compression in software size which will have an impact on the software maintenance func ,on.

Readiness for use

Although it has been used in several efforts, Modula as well as CPascal, was designed as an experiment to illustrate a concept. Therefore, Modula may lack the necessary support to be seriously considered for a project of the magnitude of a DDP BMD effort.

7.4.3. Concurrent SP/k

The University of Toronto defined a series of subsets of PL/I named SP/k, with SP/7 being the most recent one. These languages were originally defined as teaching tools, but have since then been distributed to other universities and institutions and seem to be well-accepted. The need for concurrent features in a programming language led to further extension of SP/k series. Concurrent features were added to form the Concurrent Sp/k (CSP/k) language.

Concurrent Language Features

The concurrent features of CSP/k are:

1) Processes, reentrant procedures;

2) Monitors, entries;

3) Condition variables,signaling and waiting;

4) The busy statement, simulation.

The terms here are used in a sense that is compatible with those in Concurrent Pascal.

Readiness for Use

SP/k was implemented on the IBM 360/370, Digital Equipment PDP-11, and UNIVAC 9030 computers. CSP/k has been implemented so far on IBM 360/370. The compiler was written in the high-level systems language SUE and shows that in this case concurrent features have not cost much in terms of performance. Code produced from CSP/k runs only 0.3% slower per process than code from SP/k. The addition of concurrent features to the SP/k

compiler required only about six weeks of programming effort.

Environment

Since CSP/k runs on the IBM 360/370 series of machines, all the software support tools that are available on these machines should be available to CSP/k users.

Performance

Use of CSP/k at the University of Toronto in connection with the Z7 operating system projects showed that runs of ten processes and almost one million executed CSP/k statements have required about two minutes of IBM 370/165-II CPU time. The compiler was almost error-free from the beginning.

Experience in Usage

So far CSP/k has been used as a teaching tool and has been applied to operating system development. The compiler, however, is past the experimental phase and is being distributed to anyone interested.

Conduciveness to Good Software Engineering

PL/I has been shown to e quite conducive to good engineering principles such as structured programming. CSP/k, being PL/I based, has good control structures that will allow the practice of levels of abstraction, well-defined interfaces, and more-easily verifiable programs. The compiler was also designed to have good error handling and recovery capabilities in that the generated code will automatically recover from a large number of errors.

7.4.4. Ada

Ada [Ichbiah 79], a modern algorithmic language which provides good control structures and precise control over the representation of data, is one of the first major efforts to include real-time programming facilities for modelling parallel tasks, handling exceptions, providing access to system dependent parameters, and encapsulation facilities with well defined interfaces to support good programming techniques. The language was designed with three overriding concerns:

- a recognition of the importance of program reliability and maintenance,
- a concern for programming as a human activity, and
- efficiency.

A further discussion of these points can be found in the "Preliminary Ada Reference Manual" [Ichbiah 79].

General Language Features

Ada has language features for sequential programming similar to those provided by CPascal and Modula. In addition Ada supports fixed and floating point real numbers as a basic data type. The string, a one dimensional array of type character, is also a basic data type. Pointers or access types are supported by the language. Ada control statements differ from CPascal in that they do not supply a repeat statement; however, they do supply a return statement. The addition of a return will certainly help to make programs more readable. Other features include short circuit conditions and assert statements. The short circuit provides a method of skipping past the evaluation of the rest of a string of conditions upon evaluating one which

is not true. The assert statement can be used to help insure that execution will not continue beyond the assert statement unless the stated assertion is true. Overloading, the ability of defining an additional meaning for an operator, is one of the more controversial features of the language.

As the language is intended for the production of large real-time embedded systems, it also supports good I/O and computational facilities. The package and task modules provide the basic unit of encapsulation. The interface between modules is explicitly defined and side effects are eliminated because all parameters must be defined as in, out, or in out. Defined as part of the language are facilities usually supplied by the operating system for linking separately compiled modules. The explicit definition of the interface units in an Ada module will be useful in enabling the design and implementation of good support tools. In particular, consistency checkers and intermodule type checking will be relatively easy in an Ada programming development facility.

Concurrent Language Features

An Ada TASK is a module that may operate in parallel with other Ada TASKs. A TASK consists of two parts: a TASK specification and the TASK body. The specification can specify either a single TASK or a family of similar tasks. The specifications consist of declarations specifying the interface between the task and other external units. Entry declarations in the visible specification part define communication between tasks in mutual exclusion. Declarations of variable and modules are not allowed in the visible part. The task body specifies the

execution of a task. It may contain accept and select statements as well as local entry declarations.

Tasking as defined in the Ada Reference Manual is expected to be modified in a new version of Ada due to be released on July 15, 1980. As we understand the changes, the concept of families of tasks will be replaced by allowing a task to be referred to by an access variable. While the proposed changes introduce some new problems, we feel that the capability of creating new tasks dynamically at run time allows for more realistic modeling of typical BMD applications.

Task communication and synchronization is achieved via the ENTRY declarations and the ACCEPT statements of an Ada task. Mutual exclusion to shared data is guaranteed during the execution of an accept statement in a task. The SELECT statement allows a selective wait on one or more alternatives. The DELAY statement provides a method for suspending the task for at least a given time interval.

Overall, the tasking facility of Ada appears to be sufficient to provide a method of specifying concurrent processing and the necessary communication and synchronization primitives to model a BMD DDP application.

Program Development Environment

The DoD is currently expending considerable effort to define a suitable program development environment for Ada. The Ada Language Environment (ALE) consists of the Ada Language Integrated Computer Environment (ALICE) plus additional policy and procedural issues. The following have been identified as

desired characteristics of ALICE:

1. Simplicity - The structure of the environment follows a natural simple overall concept.

2. Responsiveness - The environment provides a coordinated complete set of useful tools.

3. Open-endedness - The environment is adaptable to updates, improvements, and changes.

4. Implementability - The programs which form the environment are developed completely in Ada.

5. Commonality - The environment is maintained as part of a library available to all users.

6. Efficiency - The programs in the environment execute efficiently and utilize reasonable resources on a variety of machines.

7. Uniformity of Protocol - Communication between users and tools is uniform and involve a minimal number of different concepts and language conventions.

8. Consistency of Documentation - The ability to develop, test, and maintain environment tools is part of the environment.

9. Ease of Learning and Use - This characteristic should be inherent in the structured, coherent, and consistent design of the environment.

10. Flexibility of Use - The environment is appropriate for both batch and interactive processing.

For a more complete description of the Ada program development environment see Section 8.

Run-Time Environment

As the run time environment of Ada is still being defined, it has been difficult to find much information in this area. However, as mentioned before, the development of microcodable machines and other sophisticated hardware under the support of programs like VHSIC will no doubt have a strong impact on the run-time environment of Ada programs. Implementations of the Nebula machine will simplify the compilers and the run-time support needed for Ada or any of the other languages or operating systems which support concurrent processing.

We are aware of one effort where C, the UNIX assembly language, is being used to write a run-time system for Ada. Wand and Holden of the University of York in York, England are working on a run-time system for Ada that will run on the PDP-11/LSI-11 range of computers.

Experience in Usage

There is a translator, an interpreter, and several compilers for a subset of the Ada language available for experimental use. The test translator for syntax analysis is available via the ARPA, TELA, and TYME nets. It runs under the Multex system at MIT.

There are two compiler efforts at CMU: the TCOL Ada [Brosgol 80], and the Ada Charet effort. The TCOL Ada project is in conjunction with the Production Quality Compiler Compiler (PQCC)

[Newcomer 79]. The TCOL project is a joint effort between CMU and Intermetrics,Inc. The intention is to define an intermediate representation for the preliminary Ada language. Intermediate representations can be used for many applications:

- language-oriented editing,
- pretty printing
- generation of verification conditions,
- direct interpretation (as in a special debugging system),
- automatic test-condition generation,
- a common representation for family of retargetable compilers, and
- as a representation within a single compiler.

The TCOL Ada being defined at CMU converts a Ada program to a directed graph structure. This intermediate representation has been defined specifically for Ada and for post-semantic processing. It contains all (and only) the information needed by the "back end" of a compiler and by similar applications that require that semantic analysis have already been done.

The Ada Charet effort is aimed at producing a compiler for a subset of Ada. The object was to identify some of the implementation problems in Ada. They have a BLISS implementation of a subset of Ada running on a PDP-10 machine with a TOPS-10 operating system. The front end parser and analyzer, written in Simula and borrowed from Intermetrics, also runs on the PDP-10 but takes a large amount of space. The Charet effort does not support tasking.

The Courant Institute at New York University has written an

interpreter for Ada in SETL (an Execution Specification Language). This was aimed at identifying implementation problems with regard to the TASKing facilities of Ada. It runs on a VAX with a VMS operating system.

Readiness For Use

A revised language reference is expected to be released on July 15th. In particular, the definition of TASKING will be changed significantly. There will be a TYPE TASK, therefore one can now create TASKs dynamically at run-time.

SofTech was awarded the contract for the Army compiler to be completed in 1982.

Although considerable time and money is being spent on the Ada project, it is not quite ready for use. However, in our estimation, Ada will in the near future provide the most satisfactory vehicle for implementing large sophisticated embedded systems of the magnitude required by a DDP BMD application.

7.5. Track Management Examples

Sample programs were coded to obtain some hands-on experience in coding in a concurrent language and to better understand the languages. We have included sample concurrent programs coded in Ada and Concurrent Pascal. The example coded in Ada is taken from the "Rationale for the Design of the ADA Programming Language" [Ichbiah 79]. We have tried to represent the same example as it might be coded in CPascal.

The CPascal example is considerably longer, but its structure seems to be better. The communication between the system components and the tight encapsulation seems to be more apparent in the CPascal program. The most difficult part was the identification of functions which needed to be part of a monitor and those which best belonged in one or another of the concurrent processes.

It is our understanding that the tasking facilities of Ada are to be modified in such a way as to support tasking as a typed variable. This change would permit dynamic task creation and deletion. The ability to create and delete tasks dynamically would have been useful in this example. The coding in both Ada and CPascal required an array of track tasks to be created. This was somewhat artificial.

The realization that the system might actually be implemented as a multiprocess system caused some conceptual problems. In retrospect however, we realize the problems were a result of thinking in languages which can not provide the necessary mutual exclusion. This leads to the conclusion that

retraining of programmers will be necessary before the full advantage of a language powerful enough to support concurrent processing can be realized.

7.5.1. Ada Example

The following is an example of an Ada program taken from the Rationale for the Design of the Ada Programming Language manual page 11-25. The example shows the use of packages and tasks to realize a complex real-time system, such as radar surveillance. The track-management package introduces the abstract notion of a track. Several tracks can coexist. A current position and a current speed vector in a two-dimensional space are associated with each track. The position is updated regularly from the value of the speed. Both the position and speed can be modified externally, or examined. The restrictions are that certain values should not be read while they are being changed. We thus have a classical reader-writer problem.

The track_management package is the main or controlling program. Declared within the track_manager are two types of tasks, a single track_control task and 256 track tasks. The main purpose of the track_control task is to manage the creation and killing of new track tasks. The table track_name indicates, for each track, the unique name currently assigned to it. This table can be accessed via the track_control task and each of the 256 track tasks.

ADA EXAMPLE OF A RADAR TRACK MANAGEMENT PACKAGE

```
PACKAGE track_management IS
  TYPE track_info IS
     RECORD
       x, y        :miles;                  -global type
       vx, vy      :miles-per-second;       -global type
       t : time;
     END RECORD;

  TYPE track_id IS PRIVATE;

     FUNCTION     create_track   (init  : IN track_info) RETURN track_id
     PROCEDURE    kill_track     (t   : IN track_id);
     FUNCTION     read_track     (t   : IN track_id) RETURN track_info;
     PROCEDURE    change_track   (t   : IN track_ik; d : IN track_info);
  no_more_tracks, illegal_track : EXCEPTION;
PRIVATE
  max_track  :  CONSTANT integer  := 512;
  SUBTYPE track_range IS integer RANGE 0 .. max_track;
  SUBTYPE name_type   IS long_integer;
  TYPE track_id IS
     RECORD
       index  :  track_range
       unique_name  :  name_type;
     END RECORD;
END track_management;

PACKAGE BODY track_management IS
  null_track  : CONSTANT track_id := (0, 0);
  track_name  : ARRAY (1 max_track) OF name_type :=(1..max_track => 0);
  --this is a table indicating, for each track,the unique name currentl,
    assigned to it.
  last_name   : name_type  := 0;  -- the last unique-name used.


  PROCEDURE check_track (t  : IN track_id);


  TASK track (1..max_track) IS
     PROCEDURE read(i  :  OUT track_info);
     ENTRY change(i  :  IN track_info);
     ENTRY initialize(i  :  IN track_info);
     ENTRY kill;
END track;

     TASK track_control IS
       ENTRY create_track(id  : OUT track_id);
       ENTRY kill_track(t  : IN track_id);
     END track_control;

     PROCEDURE check_track (t  IN track_id)IS
        -- to ensure the validity of a track value:
        -- * it has a positive index,
        -- * it has the same unique name as that known to the system,
        -- * it is still active.
     BEGIN
```

```
            IF t.index = 0
               OR ELSE track_name(t.index) /= t.unique_name
               OR ELSE NOT track(t.index)active THEN
               RAISE illegal_track;
            END IF;
         END check_track;

         FUNCTION create_track (init  : IN track_info) RETURN track_id IS
            new_track  : track_id;
         BEGIN
            track_control.create_track(new_track);
            INITIATE track(new_track.index);
            track(new_track.index).initialize(init);
            RETURN new_track;
         END create_track;

         PROCEDURE kill_track(t  : IN track_id) IS
         BEGIN
            check_track(t);
            track(t.index).kill;
            track_control.kill_track(t);
         END kill_track;

         FUNCTION read_track (t  :IN track_id) RETURN track_info IS
            i  : track_info;
         BEGIN
            check_track(t);
            track(t.index).read(1);
            RETURN i;
         EXCEPTION
            WHEN tasking_error => RAISE illegal_track;
         END read_track;

         PROCEDURE change_track(t);
         BEGIN
            check_track(t);
            track(t.index) change(d);
         EXCEPTION
            WHEN tasking_error => RAISE illegal_track;
         END change_track;

      TASK BODY track IS
         data     : track_info;
         readers  : integer  := 0;
         ENTRY start_read;
         ENTRY stop_read;

         PROCEDURE read (i  : OUT track_info) IS
         BEGIN
            start_read;
            i := data;
            stop_read;
         END read;

         PROCEDURE update_position IS
            new_time  : time :=system clock;
            delta_time  : time:= new_time - data.t;
         BEGIN
```

```
        data.x := data.x + delta_time*data.vx;
        data.y := data.y + delta_time*data.vy;
        data.t := new_time;
    END update_position;

BEGIN       --body of track;
        ACCEPT initialize (i : IN track_info) DO
        data :=i;
    END initialize;
    update_position;

    LOOP
        SELECT
            WHEN change count = 0 AND kill'count = 0 =>
                ACCEPT start_read;
                readers := readers + 1;
      OR
            WHEN readers > 0 =>
                ACCEPT stop_read;
                readers := readers - 1;
        OR
            WHEN readers = 0 AND kill'count = 0 =>
                ACCEPT change (i : IN track_info) DO
                   data := i;
                END change;
                update_position;
        OR
            ACCEPT kill;
            EXIT;
        OR
            DELAY 0.10*seconds;
            update_position;
        END SELECT;
   END LOOP;
END track;

           TASK BODY track_control IS
               FUNCTION fine_track RETURN track_range IS
               BEGIN
                 FOR i IN  1 .. max_track LOOP
                   IF track_name (i) = 0 THEN
                        RETURN i;
                   END IF;
                 END LOOP;
                 RAISE no_more_tracks;
               END fine_track;

            BEGIN       -- body of track_control
               LOOP
                  BEGIN
                       SELECT
                            ACCEPT create_track(id  : OUT track_id) DO
                               id.index := find_track;
                               IF last_name = name_type'last THEN
                                  last_name := 1;
                               ELSE
                                  last_name := last_name + 1;
                               en  IF;
```

```
                        track_name(id.index) := last_name;
                        id.unique_name := last_name;
                    END create_track;
                OR
                    ACCEPT kill_track(t  : IN  track_id) DO
                        track_name(t)  := 0;
                    END kill_track;
                END SELECT;
            EXCEPTION
                WHEN no_more_tracks => NULL;
            END;
        END LOOP;
    END track_control;

BEGIN    --body of track_management
        INITIATE track_control;
END track_management;
```

Tracks are manipulated by external agents through the operations CREATE_TRACK (to start a new track, with an initial value), READ_TRACK (to obtain the current position on a track) CHANGE_TRACK (to modify track data) and KILL_TRACK (to release the track).

All tracks are independent. This is achieved by associating a particular task from a family to a newly created track. The global management of a pool of tasks is achieved by the TRACK-CONTROL task. Note that the TRACK tasks act as servers, in the sense that an activation of one task corresponds to one track, but the same task can represent different tracks in different successive activations.

In order to preserve some integrity in the way tracks are used (for example, to ensure that a reference to a track is not that of an obsolete activation), a unique name is associated with each active track. The unique name is a long integer which is incremented at each track creation, and recorded in the track identification. It acts as a sort of password,in that, for each active track, the system keeps the unique name currently

associated to it in the array TRACK_NAME. This one is checked against that contained in the track identification. Using a LONG_INTEGER for unique names should guarantee that the same name is not reused before a reasonable period of time.

7.5.2. Concurrent Pascal Example

In CPascal the track_management package could appear as a track_control process, designed as follows. The data of type track_info is defined within a monitor as it represents information which must be accessed by both the track_management process and by the track process. Likewise, the 256 element array track_name would be defined within a monitor as it must be accessed by both the track_management and by the track_control process.

```
*************************************
                FIFO CLASS
*************************************

TYPE fifo =
CLASS(limit: integer);

VAR head, tail, length: integer;

FUNCTION ENTRY arrival: integer;
BEGIN
  arrival:= tail;
  tail:= tail MOD limit + 1;
  length:= length + 1;
END;

FUNCTION ENTRY departure: integer;
BEGIN
  departure:= head;
  head:= head MOD limit + 1;
  length:= length -1;
END;

FUNCTION ENTRY empty: boolean;
BEGIN empty:= (length = 0) END;

FUNCTION ENTRY full: boolean;
BEGIN full:= length = limit) END;

BEGIN head:= 1; tail:= 1; length:= 0 END;
```

```
*****************************************
        TRACK_READ CLASS
*****************************************

TYPE track_info = RECORD
                    x, y        :miles
                    vx, vy      :miles_per_sec;
                    t           :time;
                  END;

TYPE track_read =
CLASS (track_data: ind_track_data);

VAR data_ptr:^track_info;

PROCEDURE ENTRY read (data: track_info);
BEGIN
  track_data.request_read (data_ptr);
  data:= data_ptr^
  track_date.release_read (date_ptr);
END;

BEGIN END;
```

```
*****************************************
        IND_TRACK_DATA MONITOR
*****************************************

        (* I have fudged CPascal to more closely model the example.
           CPascal as defined by Brinch Hansen does not support pointer
           types. So the reading of shared data would have to be done
           in the monitor, thus locking out other readers.  This is very
           restrictive and several of the CPascal varients support
           pointers. *)

CONT processcount = # of processes with access to an ind_track_data;
TYPE processqueue = ARRAY[1..processcount] OF queue;

TYPE track-info = RECORD
                    x,y          :miles
                    vx, vy       :miles_per_sec;
                    t            :time;
                  END;

TYPE ind_track_data =
MONITOR

VAR my_data: ^track_info;
  alive, reader, change_pending: boolean;
  read-count: integer;
  change_q, read_q: processqueue;
  next_change, next_read: fifo;

FUNCTION ENTRY active: boolean;
BEGIN
  active:= alive;
END;

PROCEDURE ENTRY activate (data: track_info);

BEGIN
  read_count:= 0;
  reader:= false;
  change_pending:= false;
  INIT next_change, next_read;
  NEW(my_data);
  my_data^:= data;
  alive:= true;
END;

PROCEDURE ENTRY kill (data: track_info);
BEGIN
  IF alive THEN
  BEGIN
    alive:= false;
    WHILE change_pending
      DO continue(change_q[next_change.departure]);
    WHILE NOT read_q[next_read.empty]
      DO continue(read_q[next_read.departure]);
  END;
END;
```

```
PROCEDURE ENTRY change (data: track_info; ok: boolean);
BEGIN
  IF alive THEN
  BEGIN
    change_pending:= true;
    IF reader THEN delay(change_q[next_change.arrival]);
    my_data^:= data;
    IF change_q[next_change.empty] THEN
      change_pending:= false
    ELSE
      continue(change_q[next_change.departure]);
    IF NOT read_q[next_change.empty] THEN
      continue(read_q[next_read.departure]);
  END;
END;


PROCEDURE ENTRY request_read (data_ptr: ^track_info);
BEGIN
  IF change_pending AND alive THEN
    delay(read_q[next_read.arrival]);
  ELSE
    IF alive THEN
      BEGIN
        readers:= true;
        read_count:= read_count + 1;
        data_ptr:= my_data;
      END;
END;

PROCEDURE ENTRY release_read (data_ptr: ^track_info);
BEGIN
  data_ptr:= null;
  IF read_count > 0 THEN
    read_count:= read_count -1;
  IF read_count = 0 THEN
    BEGIN
      reader:= false;
      If change_pending THEN continue(change_q[next_change.departure]);
    END;
END;

BEGIN
  alive:= false;
  my_data:= nil;
END;
```

```
*************************************
        TRACKER PROCESS
*************************************


TYPE track_info = RECORD
                    x, y          :miles
                    vx, vy        :miles_per_sec;
                    t             :time;
                  END;

TYPE tracker =
PROCESS (track_data:ind_track_data; clock: system_clock);

(* The track process assumes the existence of a system_clock monitor
  with entries time, alarm_set, wake. The ability to assume these
  capabilities is an illustration of how step wise refinement can be
  used when developing a program top down. *)

VAR new_time, delta_time: time; data: track_info;
  track: track_read;

PROCEDURE update_position;

BEGIN
  data.x:= data.x + delta_time * data.vx;
  data.y:= data.y + delta_time * data.vy;
  data.t:= new_time;
END;

PROCEDURE initialize;
BEGIN
  --code to initialize the clock

  INIT track (track_data);
END;

BEGIN
  initialize;
  CYCLE
    new_time:= clock.time;
    delta_time:= new_time - data.t;
    IF track_data.active THEN
      BEGIN
        track.read(data);
        update_position;
        track_data.change(data);
      END;
    clock.alarm_set(new_time + 0.10 * seconds);
    clock.wake;
  END;
END;
```

```
**********************************
        TRACK_CONTROL MONITOR
**********************************

TYPE track_control =
MONITOR

CONST track_max = 512;
TYPE track_range = 1..track_max OF integer;

CONST processcount = # of processes with access
TYPE processqueue = ARRAY[1..processcount] OF queue;

TYPE track_id = RECORD
                  index: track_range;
                  unique_name: integer;
                END;


VAR i, last_name: integer; found: boolean;
  q: processqueue; next: fifo; id: track-id;
  track_name: ARRAY[track_range] OF integer;

PROCEDURE find_track;
  VAR i: integer;

BEGIN
  found:= false;
  i:= 1;
  REPEAT
    IF track_name(i) = 0 THEN
      BEGIN
        id.index:= i;
        found:= true;
      END
    ELSE
      if i <> max_track THEN i:= i + 1;
  UNTIL found OR i > max_track;
END;

PROCEDURE ENTRY create_track(VAR t: track_id);
BEGIN
  find_track;
  IF NOT found THEN delay(q[next.arrival])
  last_name = name MOD name_limit + 1;
  track_name(id.index):= last_name;
  id.unique_name:= last_name;
  t:= id;
END;

PROCEDURE ENTRY kill_track(t: track_id)
BEGIN
  IF next.empty THEN
    track_name(t.index):= 0;
  ELSE
    id.index:= t.index;
    continue(q[next.departure]);
```

```
END;

BEGIN
  last_name:= 0;
  found:= false;
  INIT next(processcount);
  FOR i:= 1 to track_max DO
    track_name[i]:= 0;
END;
```

```
**********************************
        TRACK_MANAGER PROCESS
**********************************

TYPE track_manager =
PROCESS(controler: track_control; track:
        ARRAY [track_range] OF ind_track_data);

        (* Param is a variable length parameter list supported by the
           kernel of the SOLO operating system. The system supporting the
           track_manager process would need to use a similar technique. *)

VAR track_name:
  ARRAY[track_range] OF track_id.unique_name; trck: track_read;

PROGRAM request(VAR param: arglist; t: track_id);
ENTRY new_track, kill_track, read_track, change_track;

FUNCTION track_ok (t: track_id): boolean;
BEGIN
  IF t.index = 0 OR track_name(t.index) <> t.unique_name THEN
    track_ok:= false
  ELSE
    track_ok:= true;
END;

FUNCTION ENTRY new_track (data: track_info): track_id;
BEGIN
  controler.create_track(new_track);
  track_name[new_track.index]:= new_track.unique_name;
  track[new_track.index].activate(data);
END;

PROCEDURE ENTRY kill_track(t: track_id);
BEGIN
  IF track_ok (t) THEN
  BEGIN
      track_name[t.index]:= 0;
      track[t.index].kill;
      controler.kill_track(t);
  END
END;

FUNCTION ENTRY read_data (t: track_id): track_info
BEGIN
  IF track_ok(t) THEN
  BEGIN
    trck.read (read_data);
  END;
END;

PROCEDURE ENTRY change_track(t: track_id; d: track_info);
BEGIN
  IF track_ok (t) THEN
    track[t].change(d);
END;
```

```
PROCEDURE initialize;

VAR param: arglist; result: resulttype;

BEGIN
  -- any code necessary for initialization

  FOR i:= 1 to track_max INIT trck ????
  call ('io    ', param, result)

  (* The call to io blocks the track_manager process until a
     request is made to track_manager's enternal program request.
     The type of request will be available in the argument list,
     param. *)

  request (param, t);
END;

BEGIN initialize END;
```

```
*****************************
        INITIAL PROCESS
*****************************


VAR
  index: integer;
  controler: track_control;
  track_data: ARRAY [track_range] OF ind_track_data;
  tracks: ARRAY [track_range] OF tracker;
  manager: track_manager;

BEGIN
  INIT controler,
  FOR index:= 1 to track_max
    BEGIN
      INIT track_data [index];
      INIT tracks [index] (track_data [index]);
    END;
  INIT manager(controler, track_data);
END.
```

7.6. Implementation Language Summary

In 1975 the High Order Language Working Group (HOLWG) was formed to establish a single high order language computer programming language appropriate for DoD embedded computer systems. One of the first tasks was to analyze existing languages to determine if one of them could satisfy the previously stated requirements. An in-depth analysis of 23 existing languages was conducted and the conclusion was that no language satisfied the requirements well enough to serve as the common language. However, rather than invent a new language, it was agreed to use an existing language as a starting point. Pascal, PL/1, and Algol were offered as base language to the design teams. Four competitive preliminary language designs were initiated in August 1977. All four of the winning contractors chose PASCAL for their baseline. Eventually, the Honeywell "green" design was chosen as the final selection and is now know as Ada.

Given the time and money available for the DDP BMD study, it was impossible to complete an extensive evaluation of concurrent languages or even a subset. Given this constraint, our approach was to identify the requirements for a DDP BMD implementation language. We then considered four most promising concurrent languages to determine if they could support these requirements. Three of the four were developed in a university setting. Consequently, they have not received the type of support which would be necessary in order for an implementation language to be used in a project of the magnitute of a DDP BMD application. However, the study of the languages and projects which have been

undertaken to improve the languages have served to illustrate the concepts that a DDP BMD implementation language should support.

Therefore, our approach to Ada was to determine if Ada provided facilities to support the required concepts. It appears that Ada does provide support for the required concepts. The unfortunate fact remains that Ada is still being modified. The lack of production-quality compilers will also be a problem for the next two years. Thus, the choice of an implementation language for a DDP BMD implementation language must be a compromise.

The project could be implemented in a language, such as Fortran, which has been used and tested over time, has good compilers and good support tools. But Fortran does not support many of the features required by the application. Or the project could be implemented in one of the experimental languages, such as CPascal, variations of which do support the required features. However, the language lacks support tools and may even lack vendor support. Finally, the application could be implemented in Ada, realizing that the language and support tools are being developed concurrently with the application program. Of the three choices, we recommend the last. Even though we realize the pitfalls of such a situation, the advantage of having tools which are designed to support the features characteristic of the application and the advantage of programming in a language which has the backing of the DoD will offset many of the problems. Our second choice would be to further investigate the work being done on CPascal at TRW. This choice has the advantage of developing the application in a language having many of the features

required by a DDP BMD application. And as Pascal was the baseline in the design of Ada, many of the features provided by CPascal will be similar to those eventually provided by Ada.

SECTION 8

SOFTWARE DEVELOPMENT ENVIRONMENT

8. SOFTWARE DEVELOPMENT ENVIRONMENT (SDE)

Important to software developers, besides a programming language suitable for the application, is the environment surrounding the language. The need for tools that can ease the jobs of program entry, maintenance of program modules, system construction, debugging have long been recognized as important. One important work that has evolved from this realization is the Programmer's Workbench [Ivie 77] developed at Bell Laboratories. The Programmer's Workbench is a facility which provides an environment conducive to the development and maintenance of software to run on computers other than those on which they are being developed, i.e. target systems.

8.1. Why A SDE?

The motivating factors for the development of such a facility are many. Following are several suggested by Ivie. First, the development of a unified set of tools for one machine will be less expensive than the development of the same or similar tools for multiple target systems. Second, the workbench approach provides a more nearly uniform programming environment even across projects. This more uniform environment can reduce training and documentation costs, facilitate the development of standard policies and procedures, and enhance the productivity of programmers. The transition to new equipment provides another opportunity for realizing the benefits of program development

facility which is distinct from the target system. The upgrading of the host can be scheduled independently of any changes or improvements to the development facility. In many cases needs of the developer conflict with those of the user. A separate and distinct program development facility makes it possible to satisfy both. A feed back loop from the target system to the development facility can facilitate the validation and testing of the system. The hardware configuration necessary to support a program development system should be much simpler and less cumbersome than those typically required in a large general purpose computer suited to application programs. Finally, the separate workbench facility encourages the development of machine independent programming tools, it helps to focus the attention on the importance of the programming environment, and should provide a stimulus for the integration of programming tools.

8.2. Components of a SDE

A review of the Proceedings of the Ada Environment Workshop, Nov 27-29, 1979 provides a good overview of the current state of program development facilities. It also defines desirable characteristics of a software development environment. The software development environment should:

- have a simple overall structure
- provide a coordinated complete set of useful tools
- be adaptable to updates, improvements, and changes
- be implemented in the language in which software is to be developed
- be readily available to all users
- be efficient

- be uniform
- enforce consistency of documentation
- be easy to learn and use
- be appropriate for both batch and interactive processing

The environment should include a wide variety of capabilities to support software development during all phases of the life cycle. In addition to being easy to use, the environment should provide motivation to "do it correctly". The capabilities of the environment can be implemented as an integrated set of general purpose, development, system integration, and management tools. The following are some of the more desirable capabilities.

General purpose tools should support:

- text editing
- document preparation
- communication
- information management

Software development tools should support:

- the entering of syntactically correct programs
- program translation
- program execution
- program debugging (preferable at the source level)

System integration and composition tools should support:

- a method to compose systems
- type checking across module boundaries
- version control and version histories

- a general system library

Management tools should provide:

- facilities for system documentation
- facilities for system modification requests and histories
- control of the state of the system

The development of Ada seems to be providing strong motivation to define environment tools in terms of support for an Ada programming environment, a notable exception is TOOLPACK: An Environment for Numerical Software Development being developed at the Argonne National Laboratory for the U.S. Department of Energy [Osterweil 79]. TOOLPACK is intended to provide a environment suitable for the development of structured Fortran programs.

An overwhelming majority of the program development facilities have been or are being developed on a Digital Equipment Machine under the UNIX operating system. The UNIX Shell program, its pipes, and the tree structured files system provide a good starting point for the development of a good interactive program development facility.

8.3. Examples of Environments

8.3.1. Programmer's Workbench

The Programmer's Workbench describes the development environment at Bell Labs. As of February 1976 there were three DEC machines in operation at Bell Labs in Piscataway, NJ and one in use at Murray Hill, NJ. These machines provided the

development facilities for producing systems to run of their IBM 370s and on the UNIVAC 1110. The five basic components of the workbench at that time were job submission, module control, change management, documentation, and test drivers.

8.3.2. Development Support Machine

The Development Support Machine (DSM) [McCauley 79] is a sophisticated support capability developed at Ford Aerospace and Communications Corporation (FACC) to reduce the cost and improve the quality of software. The facility has been implemented on the Programmer's Workbench/UNIX system. The DSM provides a common, integrated environment for the specification, design, coding and documentation of software. Development personnel work in an on-line environment with DSM. DSM provides a method of specifing requirements, tracing the requirements, and then extracting test requirements. DSM has a SPECIAL processor which was used for the formal specification and verification of the Kernel Secure Operating System [McCauley and Drangowski 79]. DSM supports an FACC-developed structured English design language called HDL (for Hierarchical Design Language). The machine_readable design language HDL reduces the need for expensive flow charting. DSM supports a variety of languages including Modula. The Source Code Control System (SCCS) [Rochkind 75] facility of DSM provides for incremental version maintenance of text. DSM also provides for the construction of complex software systems, testing, and management support capabilities. The KSOS project used the compiler-compiler system (lex and yacc) provided by UNIX to build a simple parser for Modula which turned out to be important in a large number of the

other tools. Currently, over 16 DSMs are in use on projects throughout FACC.

8.3.3. MSEF

Microprocessor Software Engineering Facility (MSEF) [Voydock 79] is an integrated set of tools to support the development and maintenance of microcomputer software. MSEF as developed by SofTech is hosted on a PDP-11 computer under the UNIX operating system. The MSEF Change Control Library promotes defining, updating, and integrating parts of a software configuration, isolation of user work environments, and version control. MSEF supports the organized testing of software componets by associating test scenarios and test results with the components to be tested. Automatic change logging with a configuration audit trail is also provided by MSEF.

8.3.4. CSDP

Communications Software Development Package (CSDP) [Allshouse 79] is a coordinated set of software tools and methodologies being designed by Computer Science Corporation's (CSC) Systems Division for Rome Air Development Center. CSDP is designed to be used on a large mainframe for developing software for embedded computer systems, which normally do not have development facilities of their own. The initial application will be to support communications software developed in the J73/C dialect of JOVIAL. CSDP will be implemented on the Honeywell 6180 under MULTICS and the target will be an Interdata 8/32 configured as a switcing testbed at RADC. CSDP supplies a user interface (the SHELL), a tool kit, a tool manager, and software

database - the Project Support Library (PSL). The following is a list of tools and techniques to be available under CSDP:

1) Management Tools - a tool for tracking, the status of a project, module by module, through design, development, and testing.

- Configuration Management
- Project Planning and Scheduling

2) Design tools -

- Program Design Language (PDL) analyzers and formatters
- Design verification, by review, walkthrough, and checking against requirements.

3) Implementation -

- Compiler and linker Loader
- Debugger, source level oriented
- Structure Analyzer

4) Test Tools -

- Test tools to create data and drive systems
- Error reporting systems

5) Maintenance Tools -

- Configuration Manager, coupled to the Project Support Library
- Change requester mentioned under testing
- Mailbox system

6) General Purpose Tools -

- Text editor
- Report generator and text formatter
- Mailbox system from (5)

8.3.5. Integrated Software Support System

Integrated Software Support System (ISSS) [Bate and Johnson 79] is part of a continuing Texas Instruments corporate program to improve engineering productivity and quality. As most TI computer system projects are characterized by simultaneous development of hardware and software, ISSS must support integrated hardware/software design with special attention paid to hardware-software tradeoffs. ISSS seeks to take advantage of modern distributed processing technologies in order to place as much processing power as possible in the hands of each developer. Thus, ISSS includes a Programmer's Workstation, an intelligent keyboard/video display terminal, connected to local processors. Currently the ISSS supports transportable compilers, language dependent configuration managers and text editors, multi-tasking runtime environments and validation tools. Library management tools and source-code change control are user with interactive editors. A number of machine-level interactive debugging tools have been developed to support microprocessor applications. According to some calculations made by TI a software development facility costing slightly more than $13,000 which provides a 20% productivity improvement will pay for the equipment in six months.

8.3.6. MUST

Multipurpose User-oriented Software Technology (MUST) [Merilatt 79] is an integrated verification and testing system being designed by Boeing Computer Services for NASA. MUST is composed of the three major components: 1) a common, machine independent, user interface to all elemets of the environment, 2)

a system database which is the repository for every known about a system from requiremets through object code, and 3) a comprehensive integrated tool set which emphasizes lifecycle verification from requirements definition through maintenance. Currently, the user interface, the front-end processor for the host language (Hal/S), some target machine code generators, and an interpretive computer simmulator have been implemented.

8.3.7. MONSTR

MONitor for Software Trouble Reporting (MONSTR) [Cashman 79] is a communication-oriented maintenance system currently being built at Massachusetts Computer Associates, Inc. MONSTR is a protocol-driver system which can constrain project communication in accordance with the given protocols. MONSTR addresses the need to coordinate the activities of many people who must communicate in a variety of ways, pertaining to the status and history of a software project.

8.3.8. GANDALF

GANDALF [Habermann 79] is a host environment being developed at Carnegie Mellon University for the development of programming projects using the Ada language. GANDALF deals with the software development of a system at three levels:

- Programming issues level: those issues which arrise when a single programmer takes a program all the way from its specification to a working version.
- System composition level: the issues which arrise when a system is built by integrating many programs into one,
- Management level: pertains to issues which arrise when a

group of persons develops and maintains a system over a period of time.

The three levels are being addressed by three separate projects under the GANDALF umbrella.

INCREMENTAL PROGRAMMING ENVIRONMENT [Feiler 80]

The process of turning specifications into a working program consists of two important pieces - the methodology and the tools cycle. The programming environment tools must enable a programmer to:

- enter and modify programs
- translate, compile or interpret the programs
- link, load, and execute the programs, and
- debug the programs.

These functions are usually provided by

- the editor
- the compiler or interpreter
- the linker and loader, and
- the debugger.

GANDALF replaces these four tools by two:

- syntax editor, and
- an unparser.

These two tools work together to provide an environment where the usual process to editing, translating, linking, loading, and debugging programs does not require the ususal process of moving among tools to perform the required tasks. Such a system should serve to reduce the software development

time. These two tools are now provided for GC, a type-checked variation of C that runs under UNIX operating system on a DEC VAX.

INTERCOL [Tichy 79]

The system composition environment is concerned with the integration of many subsystems into a version of a larger system. Belady' "Law of Continuing Change" says that all large software systems are subject to modification over their entire lifespan. This leads of problems pertaining to interface control and to version control. A prototype implementation of the interface portion of INTERCOL, a language for programming-in-the-large and describing version control of a system, has been implemented on the UNIX system at CMU as a part of the GANDALF project.

A system descriptor consists of four parts:

- the provide list,
- the require list,
- the components, and
- the list of composition

An important goal of system construction is the checking of type information across module boundaries. This is also a goal of Ada. An environment such as the one provided by GANDALF would support such a goal.

GANDALF

GANDALF takes a technical approach to creating a management environment. This environment consists of two parts -- one that

is responsible for coordinating the state of the system and another that is responsible for the generation and proliferation of documentation. The state information portion is called Software Development Control (SDC) [Notkin and Habermann 78]. SDC is implemented as an extension of UNIX running of a PDP-11. As part of the evaluation of Ada, the SDC system has been coded in Ada. The Ada version is not available yet, however, a version is available for distribution including a user's manual and a standard UNIX manual for each SDC command. The SDC commands amount to roughly 1k lines of C-code.

SDC supports the development and maintenance of systems by maintaining modification histories and standard documentation and by applying tests when source objects are replaced.

8.4. Software Development Environment Summary

The concept of a system development facility and the introduction of Ada provide the opportunity to define a system development environment which can revolutionize the industry. The availability of a language designed for the development of embedded systems and an integrated set of tools which facilitate the system development process from the requirements to the test and maintance phases, can significantly shorten the development time and therefore reduce the costs.

Given the time and money available, we were unable to evaluate tools which are being developed for the system development environment. However, we have identified two of the more revolutionary to be the syntax directed editors and the source level debuggers.

Other observations include the realization that support for tools which attempt to deal with system development issues must have been considered in the design of the Ada language. Also, the UNIX operating system is especially well structured to support program development, system integration, and many of the testing and management tools currently being developed for the Ada environment.

SECTION 9

SIMULATION AND EMULATION

9. SIMULATION AND EMULATION

This section will deal with simulation and emulation, important techniques applicable to various stages of systems design and development. In the context of this section, simulation is achieved by software and is discussed as applied to high level network design as well as hardware design. Emulation is achieved by hardware, and is discussed as applied to the design of processors. The use of an emulator as a debugging tool is also suggested. Finally, the simulation and emulation of multiple processors are presented.

9.1. Computerized Simulation Modeling

Computerized simulation modeling has proven to be an effective tool for the analysis of complex, interactive systems such as distributive processing systems. It has been used to study the behavior of both existing and theoretical systems. Simulation is basically an integrative tool for coping with the complexity which in a distributive system stems predominantly from interconnections and restrictions in the interconnections among the processes.

A simulation model provides a number of benefits to a developing software system. First of all, it permits a better understanding of the system and the environment in which the system operates. Initially it can play an important step in the validation of the system requirements. By building a simulation

model, the consistency and completeness of the system specification can be checked. Savings in system development time would result if errors in the specifications were detected before the system was actually built. The detailed information needed to completely define the system would be indicated in the construction of the simulation model.

Secondly, it provides a framework for testing the desirability for system modifications. A simulative model affords the opportunity to construct ideally configured systems and the best operating conditions on the basis of the results obtained from the model. Modifications that perhaps lead to better systems could be tested on the model in order to determine their effect. A simulation model is easier to manipulate than the actual system and thus provides a practical way to evaluate alternative system design.

A simulation of a system also permits control over more sources of variation than the direct study of a system would allow. The complex interaction of the processes can thus be studied in detail so that possible predictions can be made about the system in terms of time or events.

And lastly, simulation models enable an analysis of a system to be done in a shorter period of time. All of these benefits from models help to improve the system performance and reduce the cost of designing and constructing the system.

Computerized simulation models fall basically into one of two major categories. Continuous, where the interest is in smooth changes over time in the system attributes (often

characterized by sets of differential equations) and discrete, where the cause and effect of individual discrete events are of prime interest. In discrete models, time is advanced by either moving in a discrete time increment or by moving directly from one event to the time of the next scheduled event. In a continuous model, time is advanced in very small increments.

Basically the steps needed in developing either a continuous or discrete modes are:

1) Definition of the system and its environment,

2) Data collection for the system,

3) Construction of simulation programs,

4) Validation of the model,

5) Design and execution of experiments, and

6) Analysis of results.

One of the first steps in model development is the choice of a programming language in which to code the simulation model. A general discussion of requirements for a language used in simulation modeling is given in the next section, followed by a brief description and discussion of the most popular simulation languages.

9.2. Language Requirements

The choice of an appropriate language in which to code the simulation model is a very important one and a number of alternatives should be considered. Simulations are often

programmed in general purpose programming languages such as FORTRAN, PASCAL or PL/1. However, these languages require an intensive programming effort in order to evaluate the components of a system and their interactions. A number of programming languages exist for the express purpose of simulation and they generally require less programming than the general purpose languages. Thus the major advantage of using a simulation language over other languages is the reduction of programming time and effort required to program the model. The major disadvantage is that the programmers might have to learn a new language.

It is important to have a number of features built into the language that are needed in the design of distributed processing system, and these include:

1) Parallel or pseudo-parallel processing with possible multiple activations of a given process,

2) A method of time keeping and scheduling of events or actions,

3) A method of sharing of data and resources with the restrictive mutual exclusion,

4) Special statistical accounting or reporting capabilities,

5) Generation of sequences of random numbers with various distribution,

6) Software support tools.

Languages vary widely in the ability to provide these features. Simulation of distributed systems are generally complex and difficult to understand and program. There is a great need for software support tools in the language. It is desirous to have a language that provides for modular construction of the model. A common principle in programming larger software systems is decomposition of the system into a set of smaller components which are either defined or decomposed further. Security of information and the capability to restrict access to information also promote the concept of modularity. A modular program is easier to write, understand and modify. Thus, the structure in the language that lead to modular construction of a model are extremely important.

Good debugging tools and error diagnostics, both compile time and execution time, are also important considerations when choosing a language. Debugging facilities should include such functions as setting break points, continuing with execution, displaying execution state and modifying variables. Another language tool that has recently been receiving attention is that of documentation. There is a need for some documentation tools (forced documentation being an ideal). An interactive language also has some distinct advantages, such as flexibility and greater user control but there are also some disadvantages, such as execution cost.

Thus, there are many language features to consider when choosing a simulation language. However, the decision as to which programming language to choose also depends on other factors such as

(i) language availability,

(ii) language adequately supported,

(iii) cost to obtain, install and update language,

(iv) difficulty in learning language and

(v) execution time efficiency

Simulation languages vary greatly in the capabilities they include. A discussion of the 4 most popular simulation languages follows. These languages were chosen because they have been used in many simulation models, they are generally available, and they do provide many of the desirable features. Each has certain strengths and weaknesses when considering a distributed processing system.

9.3. Simulation Languages

Due to the reduced programming effort on using simulation languages, general purpose languages will not be considered. Languages used for both continuous modeling and discrete modeling are included. The discrete languages themselves are divided into flow-chart oriented and statement-oriented languages. In the first case, the user defines flow chart blocks and then converts them into a program structure. The statement-oriented languages use program statements to define conditions that must apply before certain actions can take place.

9.3.1. GASP IV [Pritsker 74]

GASP IV is a Fortran-based simulation language which can be used in simulation studies of discrete, continuous and combined systems. It provides a user with a collection of FORTRAN programs that perform functions useful in a simulation setting such as program monitoring, error reporting, data collection, time advancement and random number generators.

A statement-oriented language, the basic entities of GASP are events. A GASP program consists of two parts: a user part and a GASP part. The user part contains subprograms for initialization, definitions for state variables, event code definitions, condition defining events and event processing. The GASP port contains subprograms that provide for such facilities as the execution controller, data and event initialization, data storage and retrieval, statistics and error reporting.

There are two different kinds of events in GASP. One is time-event and the other is state-event. Time events are those which occur at a particular time in the simulation. Those which occur when the system reaches some particular state are called state-events.

GASP is a fairly efficient language. However, being Fortran-based, it does not provide features particularly viable for modularity. The underlying storage management scheme in GASP is fixed which means that all storage for the system must be allocated at the beginning of the execution. This is a severe restriction when modeling a large complex system which is by nature dynamic and in which the number of possible events are not

predetermined.

9.3.2. GPSS [Gordon 78]

Unlike most of the other simulation languages, GPSS programs are based on a block diagram drawn by the user to represent the system to be simulated. It is then both a language and a computer program. The language is used to describe the model and the computer program is designed to accept a model described in the language and simulate the system being modeled.

GPSS has a process oriented viewpoint. Its design is based on the fundamental assumption that most systems can be simulated adequately using just the entities transactions, equipment (acted upon by the entities), and blocks which specify the logic of the system. Other elements are provided for statistical measurement. A user of GPSS defines the components of the system in terms of the entities and tranfers a flow chart representation of a dynamic system into the GPSS Block Diagram. The Block Diagram is used to route transactions as they move through the model. The concept of time-control is also built into the GPSS system.

GPSS does have fairly good debugging tools and tracing facilities. The overlying storage management scheme is dynamic which permits storage to be allocated when needed and reclaimed when it is no longer needed.

The most significant disadvantage is the cost of executing a GPSS program. As it is an interpretive system, the cost is fairly expensive. GPSS appears to be easier to learn than most statement-oriented languages, but it is less flexible. The more complex the model being simulated , the more difficult it is to

represent it using a Block Diagram. This is particularly true when there are complex interactions among the components. Thus, it appears to be more difficult to program complex systems in GPSS. Another disadvantage is that relative large amounts of memory space is needed for running a program in the Dec-10 implementation, even the most trivial kind of program needs about 20k.

9.3.3. SIMSCRIPT [Diviat 68]

SIMSCRIPT is a Fortran-based, computer language specially designed for use in simulation. It is a statement-oriented language based on the concept that a simulation system can be descibed by a series of entities which have attributes and the owner-member set. The status of the simulated system is modified through the execution of events. Each event is a SIMSCRIPT routine which simulates the activities of the system. There is a timing routine which keeps track of a system clock. A priority ranking also is available.

The underlying storage management scheme is dynamic so storage can be allocated and reclaimed as needed. It is a fairly efficient language. However, due to being Fortran-based, good modular structures are absent in the language.

9.3.4. SIMULA-67 [Dah168] SIMULA-67 [Dah168]

SIMULA-67 is an Algol-based discrete simulation language which extends Algol to include the concept of a collection of programs called processes conceptually operating in parallel. The concept of a process is derived from the class structure which is a coroutine control structure. Processes can be

inserted and removed from an event-time queue. A process possesses its own local data. A process can be in one of four possible states: active, scheduled, suspended and terminated. A user can activate a process directly or place it on the time event list which would then activate the process at a time determined when the process was put on the list. SIMULA is capable of handling a number of systems executing logically in parallel.

SIMULA does provide good facilities for modular program development. Modules in SIMULA can be procedures, functions, classes, blocks or processes. A class object can act as a data abstraction facility without the ability to monitor use. The DEC-10 implementation of SIMULA has the concepts of "virtual" and "hidden" variables which are in the modular construction of the model.

An interactive debugger SIMDDT is available and can provide such functions as break points, modification of variable values, display of execution after a break point.

It is a powerful language that is somewhat difficult for a novice programmer to learn. The SIMULA compiler is a large piece of software that needs 24k to execute on the DEC-10 system.

Because of the structuring facilities in the language, programs written in SIMULA are clear and fairly easy to understand and thus modify.

9.4. Simulation Modeling Verification

Regardless of the language chosen in which to code the model, the next very important step after the model construction is the verification of the model. Verification of the simulation program involves ensuring that the computer program correctly represents the intended model. There are really two parts to ensuring that the program is a correct representation: that of verification which seeks to show that the computer program performs as expected and validation which attempts to establish that the model behavior validly represents the system being simulated. The verification and validation of a model are crucial but difficult tasks often requiring compromises.

The best way to validate a simulation model is to compare the results of the model with the results of the real system under the same conditions. The results of the simulation model should be compared statistically with the results of the actual system to test for significance. Reproducibility should also be shown if possible. However reproducibility is sometimes difficult due to the statisical nature of inputs to the model.

The next best way to validate the model is to compare the results with historical data collected from the active system that was run under similar conditions. If this is not available, validation of the model by experts in the field will increase the credibility of the model.

Traditional tools for the verification of a program usually consists of run-time error messages and debugging facilities including such features as traces and dumps. Some advances have

been made in formal proofs of correctness to verify a program. However, because simulation applications are typically large with complex interactives, it appears doubtful that formal techniques have much applicability in simulation program verification. Although not guaranteeing absolute proof of correctness, complete testing of the program aids in the verification. It is recommended that a test plan and test data be chosen and specified during the design state, thus ensuring that the tests are based on system specifications rather than code. Another approach which can be used in establishing the reliability of programs is to divide the testing phase into two parts: module or unit testing where each module is exercised alone and integrative testing where the individually tested modules are merged and tested as a whole. Thus, through program traces, debugging and testing as well as using structured top-drawn design in the construction of a program, verification can be approached.

9.5. Summary

Simulation modeling plays an important role in the analysis of large, complex systems. Simulation modeling leads to improved system understanding and also indicates the amount of detail actually needed to study the system. It expedites the speed with which behavior of a system can be analyzed. It provides a framework for seeking to improve the performance of a system by various modifications in the design. A simulation model is less costly to build and permits more control over components than the actual system.

The chore of the programming language for the model is an important one. We surveyed 4 widely used simulation languages

which can be used to model distributed processing systems. Thus basically all had the concept of time control and logical parallelism. None had mutual exclusion built in as a feature and thus the model builder would have to provide this feature. The amount of debugging tools varies among the languages and their implementation. None of the languages had any good documentation features.

Because of its features for writing modular programs and the easily understandable language concepts, SIMULA 67 is the recommended language to use in modeling distributed procesing systems. It is believed that the complex interactions among processes can be be described in SIMULA.

Certainly an important part of any simulation model is the validation and verification of the model. Techniques such as comparison of results obtained form the simulation model with actual results should be used to help validate the model. Systematic and modular construction will aid in the verification.

9.6. Emulation

The most obvious use of emulation is to provide the capability of developing software for a target processor concurrently with or before the development of the processor. It is assumed, of course, that the processor is completed specified at the functional or register transfer level so that the instruction set available to the programmer is defined.

Concurrent development of hardware and software is important in distributed systems. The 'software first' approach was adequately demonstrated by the experience of the Martin Marietta Viking Lander Flight System [Wachs 78]. The QM-1 was used to emulate the flight computer which was delivered two months before launch while the actual real-time flight program was developed one year ahead of computer delivery.

9.6.1. The QM-1 Universal Emulator

The QM-1 universal emulator is a computer manufactured by Nanodata Corporation in Buffalo, NY. Unlike most other mainframe computers manufactured by corporations such as Digital, IBM, and Perkin Elmer, the QM-1 is designed solely to act as a chameleon - to mimic, (or emulate) the functions of other computers. It is an extremely flexible tool, and can emulate any processor previously or currently built, and can probably emulate anything that can be developed in the next few years. Since the QM-1 is not a simulator (that is, one computer programmed to act like another), but is instead an emulator (a computer whose hardware is dynamically modified to be another), the QM-1 is a very fast processor. Current simulators run 5,000 to 100,000 times slower

than the machine they are simulating. The QM-1 emulator runs only 2 to 100 times slower than the machine it is emulating.

Programming the QM-1 entails writing an instruction decoder/interpreter in microcode, or if one is willing to sacrifice ease of programming for speed, one can program the QM-1 in a much lower level nanocode. If one takes this latter course, the QM-1 must be used in standalone mode. Debugging the nanocode requires hands-on contact with the machine. If microcode alone is to be used, the QM-1 can be used in standalone mode, with its own disk, card reader, terminal and line printer, or it can be configured as a peripheral to another system through the use of a special interface board. If it is used as a peripheral (as has already been done with the DEC system-20), the QM-1 uses the terminals, disk storage, line printers, and terminals of the host machine, reducing the cost outlay required to get the initial system online.

Since the QM-1 essentially emulates an instruction set, it can be used as a fast back end of a simulator. One example of this is a P-machine. Currently, most PASCAL compilers generate P-Code, instead of generating directly executable machine instructions. There is then a second program which simulates the actions of the P-machine and executes the P-Code. This makes PASCAL an effectively interpreted language, which is correspondingly slower than a directly executed machine. The QM-1 can be programmed to emulate the actions of the P-machine, and execute PASCAL code at a much greater speed.

A second example is with ISPS [Barbacci 77]. Currently the ISPS simulator programs generate and execute RTM-code, for a

mythical RTM (Register Transfer Machine). This RTM is well suited for simulating arbitrary hardware systems, but it is simply a simulation itself in the current state. The QM-1 can be programmed to emulate the actions of the RTM, and execute ISPS descriptions at a much higher speed than is currently obtainable.

A system similar to the ISPS system has been developed by USC/ISI for the SMITE computer description language. This system utilizes the QM-1 as a peripheral device to a DEC system-20, permitting multiple users to run SMITE emulations. The SMITE compiler generates code which is directly executable on the QM-1 system for fast emulation of target architectures.

9.6.2. Emulation As a Debugging Tool

Besides improvement of several orders of magnitude in speed, emulating a proposed machine architecture described in a language such as ISPS or SMITE can be an effective debugging tool. The functions normally provided by a simulator, such as break points and tracing can now be provided by emulation and will therefore execute faster.

Another reason for the importance of this approach is that some target machines, even when available, may be incapable of supporting software development because of the following factors.

1) rapid proliferation of small or special purpose computers;

2) lack of raw power or memory;

3) lack of suitable peripherals;

4) lack of software support.

Emulation with debugging facilities then provides an effective way of developing software for such target processors regardless of whether they are available or not.

The benefits of providing debugging functions by emulation have been illustrated by experiences at Dahlgren Laboratory [Flink 75]. The QM-1 was used to emulate the Trident submarine processor of the order of complexity as the IBM 360. The emulation software required 8 - 11 K of control store, and 256 - 512 words of nanostore. The effort took 24 man-months, with three programmers experienced in QM-1 and three who had only cursory knowledge of QM-1.

9.7. MMPS simulator system

While there exist a great many design and debugging tools for uni-processor systems, such as simulator systems, as well as off-the-shelf in-circuit emulators for almost any microprocessor on the market, there exist few real design or debugging tools for multi-processing systems. While it is relatively easy to simulate any uni-process machine on almost any other, there is no easy method of simulating the actions of multiple processors. And while it is possible to hand code a multi-processor simulator, this is certainly not a desirable method. Any design modifications must be incorporated into the current simulator by reprogramming (possibly large) sections of the code. A network simulator cannot limit itself to the simulation of a specific network, with fixed characteristics and configurations. Rather, what is needed is a software system development tool which is a

general purpose network simulation package to provide a simulation environment for the design, checkout, and maintenance of multiprocessor computer networks.

What we have set out to do is to construct tools for the network designer. A multi-microprocessor design language and dynamically reconfigurable multi-microprocessor simulator (MMPS) [Klein 79] provides an aid to an efficient design methodology.

The simulator will have the capability to provide both hardware and software level breakpoint tracing, as well as a static method of tracing the overall system performance, for later evaluation and possible redesign. This systems is being written under the UNIX operating system, and requires the multi-process and inter-process communications features provided by UNIX. Other possible operating systems which could support MMPS implementations are VAX-VMS, VAX-UNIX, and TOPS-20 for the DEC system-20.

9.8. Emulation of Multiple Processors

While MMPS simulates multiple processor architecture, there is still a lack of tools that can emulate multiple processors. The BUCS system at Mellon Institute, Carnegie Mellon University (CMU) is a reconfigurable emulation system for multiple processors. BUCS-I [McConnell 79] allows software to dynamically change interconnections among microcomputer to evaluate or verify design and algorithms. Despite the importance of the concept, the first prototype implemented is modest in processing power and cannot meet the needs of an application such as BMD as a production tool. Rome Air Development Center (RADC) has designed

Multiple Micro-processor System (MMS) as part of their system architecture evaluation facility which is the result of a ten-year continuing effort. MMS has not yet been implemented. E. D. Jensen has also proposed to build a multiple processor emulation facility at CMU. This effort is a sizable one, and will not be available before three years [Jensen 80].

The need for such emulation in the BMD application is not clear at this point. Simulating networks aspects of a system with simulation languages or an approach such as MMPS may suffice.

SECTION 10

A BMD DDP SOFTWARE DEVELOPMENT FACILITY

10. A BMD DDP SOFTWARE DEVELOPMENT FACILITY

Based on work done so far in analyzing BMD DDP needs in the few areas selected for study, a BMD DDP software development facility is partially defined in terms of both the software and hardware capabilities (see explanations in Section 2). Figure 5 gives an overview of this first-cut definition. Each logical module is briefly discussed.

Emulator

An emulation processor will be microcoded to emulate hardware description language (ISPS or SMITE) representations efficiently. Debugging facilities will also be provided by microcode in the emulation software. This provides a flexible way of achieving emulation of an unknown target processor. When a target processor is selected, it needs only be described in the hardware description language to be emulated, and software for it can be developed. Assuming that a hardware description language is used to describe the hardware during design, emulation can take place almost immediately after the choice of a processor is made. In the conventional approach, the microcode for emulation as well as debugging has to be developed after the processor is selected. This proposed emulation facility will also provide a means of checking out the design of a candidate target processor.

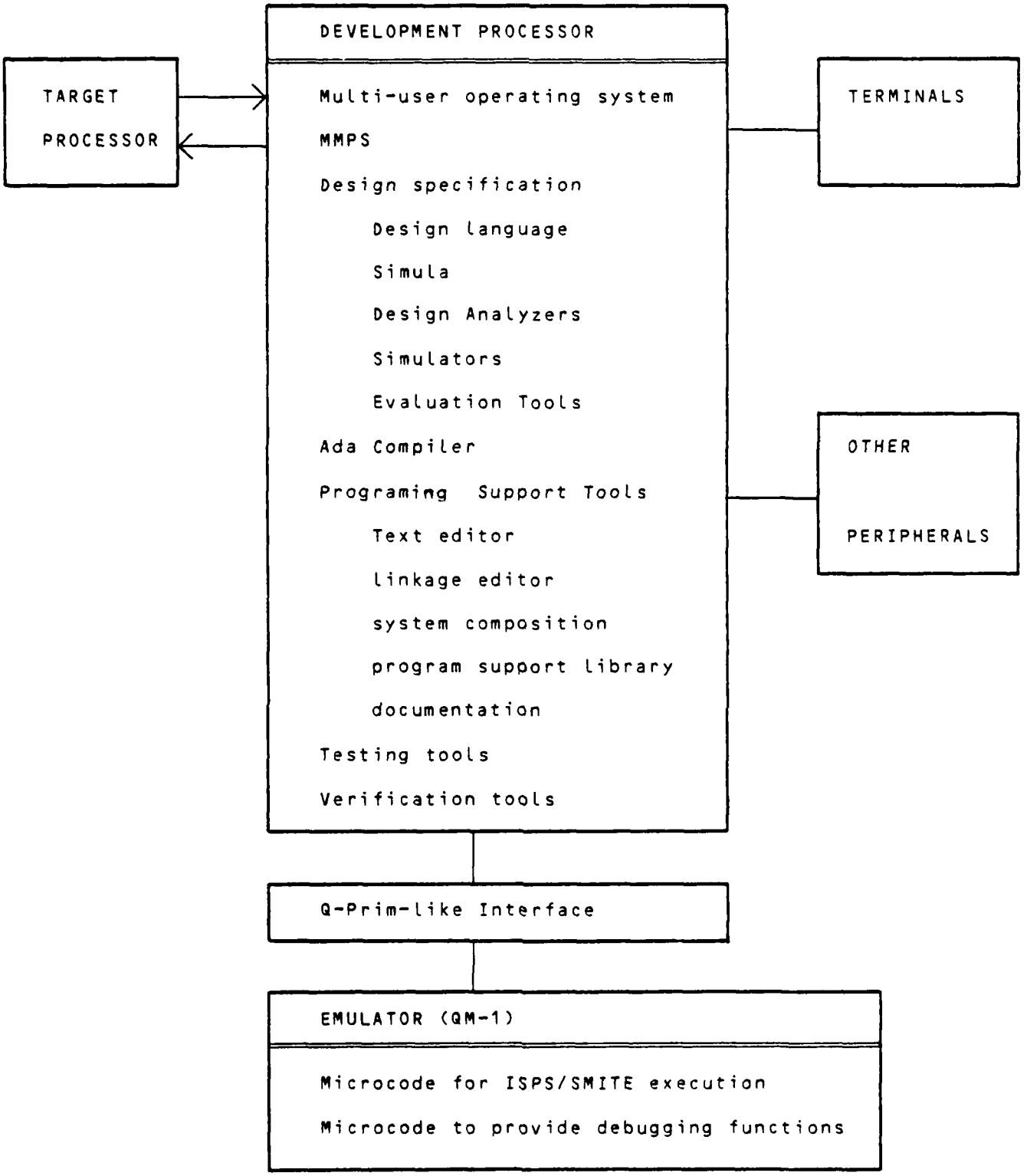


Figure 5. A BMD DDP Software Development Facility.

The QM-1 seems to be the only general purpose emulation machine available at the moment that has the power and established user-experience to warrant consideration. The other small-scale microcodable machines can be used as emulators but will not be able to provide the debugging environment so critical to software development.

Q-Prim-like Interface

To use the QM-1 in a stand-alone mode has two disadvantages for software development. First, peripherals will have to be acquired for it. This would mean higher cost, and probably duplication of peripheral between the emulation processor and the general purpose development processor. Second, processing has to be physically moved from one machine to the other, leading to fragmented capabilities not convenient for the user. A good development facility should provide an integrated set of tools with easy access to the user and easy transition from one tool to another.

It is proposed therefore that a QM-1 be connected to the main processor. There exists now a Q-Prim package that would allow a QM-1 to appear as a peripheral to the DEC-10 or DEC-20. If another processor is chosen for the development facility, Q-Prim has to be adapted.

Development Processor

This is the main processor in the development facility, supporting the tools specified in Sections 6, 7, 8, and 9. Promising candidates seem to be PDP/11, DEC-10 or -20, and VAX. PDP/11 is widely used but is comparatively more modest in

processing power than the other two. DEC-10 and -20 are efficient general purpose processors, well-established, and more cost-effective than the IBM 360/370 series, and can support the existing Q-Prim package, Simula, as well as the Air Force Ada compiler to be developed. The VAX is a newer machine, with a large instruction set, good addressing space and processing power. Most of the PDP-11 tools, though, should be easily adapted for execution on the VAX. It can also support the first full-scale Ada compiler contracted by the Army to be completed in 1982.

Choice of an operating system for the development environment depends on several factors, some of which have been discussed previously. More important considerations are that it has to support multiple users, linkage of modules compiled from different programming languages, and the MMPS (Multiple Microprocessor Simulation) concept. A multiple-user time-shared system, with a QM-1 as a peripheral, will also allow the emulation capabilities on the QM-1 to be time-shared among multiple users. On the PDP-11, either RSTS or the recently released RT-11 version 4 will support multiple users. UNIX has many good features and runs on the PDP-11 but is not commercially supported to the extent of wide-spread use. DEC-20 has the TOPS-20 operating system which supports multiple users, so does VMS on the VAX.

The ratio of using one QM-1 to one general purpose processor is not fixed. If the nature of the workload during development justifies a different mix, two QM-1s may be used as peripherals to one development processor or vice versa. However, both of

AD-A091 241 CARNEGIE-MELLON UNIV PITTSBURGH PA MELLON INST OF SCIENCE F/G 9/2
DDP DEVELOPMENT FACILITY CAPABILITIES FOR EFFECTIVE ACQUISITION--ETC(U)
JUN 80 L L CHENG, M A KOWALSKI, D V KLEIN DASG60-80-C-0016
UNCLASSIFIED NL

END
DATE FILMED
1-13
DTIC

these choices require a more complicated physical connection as well as a more complicated operating system to control. The effort and lead-time required to development software for control may not be cost effective in terms of money and time.

Terminals

Terminals are required to permit many programmers to share the processors interactively. An interactive environment is desirable for faster and more flexible user control. The number of terminals is yet to be determined. Based on the assumption that the BMD DDP subsystem is comparable to a one million Fortran instruction effort, (about 60 programmer-yrs), to be developed in two years or so, a minimum of 30 terminals are required. In addition, graphics terminals may be required to support design tools. Intelligent terminals can also be considered to relieve the main processor of some local functions such as editing. The number of users to be supported at one time may overload the main processor. In that case, the development facility can be replicated. The additional cost is not out of hand because the development facility is not duplicated as many times as the target processor (over 200). Furthermore, replication also provides back-up for reliability.

Interface With Target Processor

Some development facilities have a link available for connecting the target processor when it is delivered. This provides a means of down-loading programs already developed from the main processor to the target processor. It also minimizes the software needed to test the target processor.

Other Peripherals

Peripherals such as disks, printers, will also be necessary. The number of peripherals required should be investigated according to BMD needs.

APPENDIX A

CONCURRENCY IN FAST FOURIER TRANSFORMS

11. APPENDIX A: CONCURRENCY IN FAST FOURIER TRANSFORMS

Real-time processing has long been an objective in the area of signal and image processing Earlier real time systems were mainly analog signal/image processing systems. Efforts in achieving real-time operation using digital processors were greatly aided by the Fast Fourier Transform (FFT) algorithm. However, at the present time, real-time digital signal/image processing objectives are still unrealized with a sequential uni-processor, unless tradeoffs are made by using small sample sizes or using special-purpose hardware.

In general, it can be said that there are 5 general structural factors that affect the processing speed as well as its cost-effectiveness. They are:

(1) Hardware technology

(2) Algorithm

(3) Data size and structure

(4) Programming language

(5) Architecture

While a great deal of efforts have been made in the uniprocessor regarding these factors, advances in technolohy, multi-procellor and concurrent programming language will make it possible to increase the processing throughput. In illustrating

the potential, we will use the FFT algorithm, which is a fundamental tool in the digital signal/image processing area, to demonstrate how a concurrent programming language may be used.

The evaluation of complex Fourier coefficients in the frequency domain is a fundamental task in the digital signal and image processing. It has been a subject of intensive activity during the past fifteen years. The principle of the algorithm is well documented in most of the standard text dealing with system theory, circuit analysis, signal or image processing or communication theory. 8-point FFT algorithms using a signal flow graph can be found in chapter 6 of "Digital Signal Processing" by Oppenheim and Schafer, Prentice-Hall. signal flow graph.

In general, FFT consists of M stages of N/2 butterflies where N=number of samples and $M=\log_2 N$, and each butterfly operation may be described by the well-known butterfly equation:

$$X'(j) = X(j) + W(n) * X(k)$$
$$X'(k) = X(j) - W(n) * X(k)$$

Thus a digital processor is mainly concerned with two tasks:

(1) complex arithmetic operation and

(2) indexing and control.

Concurrency of operation may be explored at different levels:

(1) Allow concurrent operations within each butterfly, but butterflies will be operated sequentially.

(2) If M hardware-butterflies N processors are available, the algorithm may be implemented by a horizontal pipeline. In such a pipeline, each processor will sequentially compute N/2 butterflies, and then passes the results to the next stage and ready to accept data from a previous stage processor.

(3) If (N/2) processors or hardware butterflies are available, all (N/2) butterflies in one stage will be done concurrently. The array results will then be passed onto the input side of the same (N/2) butterflies to achieve a pipeline effect by software.

(4) If there are M.N processors or hardware butterflies available, the entire array may be processed through a hardware pipeline. The throughput rate would then be equal to the input rate.

APPENDIX B

HIGH ORDER LANGUAGES FOR OPERATING SYSTEMS

12. APPENDIX B: HOL FOR OPERATING SYSTEMS

Operating system programs are generally the most frequently executed programs in a computer installation. They must be designed and implemented in such a way that their demands on system resources, notably time and space, are minimized.

Because of the greater efficiency of machine level languages in computer resource utilization, most system programming has been done in machine level language. However, personnel costs are becoming the dominant cost component in computing, and this trend is likely to continue. It is expected that hardware costs will decline at an average rate of 15% per year. Personnel related costs are expected to increase at an average rate of 4% per year [Emery 78].

In 1972, a Rand Corporation study forecasted that by 1985 software will consume 95% of our defense computer system dollars. [Lieblein and Martin 79]. As hardware cost becomes smaller, the rate of increase of total cost begins to approach the rate of increase of personnel cost. In other words, it may be cheaper to buy more memory and program system components in high level languages.

Fosdick [Sigplan Notices July 1979] presented a good historical overview of the use of high-level languages for operating systems development. He points out that it was [Wirth, 1968] who started the movement towards using higher level

languages for OS programming with his report on PL360. PL360 is a high-level assembly language for the IBM series 360 computers. Although most system programming applications have been and still are implemented in the assembly language of the machine, the following are examples of exceptions to this practice.

1) PL/I was used to construct the MULTICS [Corbato] operating system wich was designed at the Massachusetts Institute of Technology in the mid-sixties. MULTICS had as one of its design techniques the use of a high level language for all machine independent functions,
2) C, a compact high level language which includes primitives to take advantage of hardware features, was used to write the UNIX [Ritchie and Thompson, 74] operating system developed at Bell Laboratories. UNIX has been implemented of the PDP-11/40 and several other small machines.
3) SAL [Tanenbaum, 74] a typeless GOTO-less language intended for system programming, was used to construct a general purpose time sharing system for the PDP-11/45.
4) BLISS [Wulf] was developed at Carnegie Mellon University. It was designed for developing operating systems for the DEC-10 machine.
5) TOOPS [Czarnik et al., 73] is the language used to write several versions of the SUE operating system.
6) Concurrent Pascal [Brinch Hansen, 78], an extension of sequential Pascal, has been used to write the Solo operating system and several other small operating systems.

In order to take advantage of the proliferation of

asynchronous hardware components available in most modern computer installations, an operating sustem must be a very "special" piece of code. It must be capable of handling some complicated conditions. But like any program, the cost of developing and maintaining an operating system is directly related to the size, complexity, and sophistication of the operating system. As operating systems became larger and more complex, it became cost effective to develop these programs in a high level language. However, the designer of a high level language for writing operating and other real-time control systems must resolve a multitude of complex issues. The goals inherent in the design of a complex real-time control system are often in direct conflict with the goals inherent in the design of a good high level language.

13. BIBLIOGRAPHY

Allshouse Richard A.
"CSDP as an Ada Environment"
Proceedings of the Ada Environment Workshop, Harbor Island, San Diego, CA
November 27-29, 1979

Barbacci M.R.,
"The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator"
Technical Report, Department of Computer Science
Carnegie-Mellon University
1977

Barrett A.T.
"An Assessment of the Application of Top Down Design and Structured
Programming,"
MITRE Working Paper WP-20830
The MITRE Corporation, Bedford, MA,
June 1976

Bate Roger R. and Douglas S. Johnson
"Texas Instruments Integrated Software Support System"
Proceedings of the Ada Environment Workshop, Harbor Island, San Diego, CA
November 27-29, 1979

Boorstyn R. R., M. Schwartz
Computr-Communication Networks = An Introduction
Seminar Notes,
Polytechnic Institute of New York,
1975

Brinch Hansen P.,
The Arcitecture of Concurrent Programs,
Prentice-Hall Inc., Englewood Cliffs, NJ
1977

Brosgol B.J; J.M. Newcomer; D,A. Lamb; D.R. Levine; M.S. Van Deusen; W.A. Wulf
TCOLAda: Revised Report on An Intermediate Representation for the Preliminary Ada Language
Internal Documentation, Carnegie Mellon University,
Computer Science Department
February 15, 1980

Cashman Paul M.
MONSTR: The NSW Tool to Monitor Software Trouble Reporting
Massachusetts Computer Associates document CA-7907-0911
September 1979

Cheng Lorna
"Program Design Languages - An Introduction"
Technical Report MTR-3446, MITRE Corp.
1977;
ESD-TR-77-324, Electronic Systems Division, MA
Jan. 1978

Clark N. Bruce
"The Total System Design Methodology"
RADC Technical Report
Rome, New York
1979

CONLAN Progress Report of the Working Group of the Conference on Computer Hardware Description Languages
Oct. 1977

Constantine L. L.,
Fundamentals of Program Design,
Prentice-Hall, NJ.,
1967

Corrigan A. E.,
"Results of an Experiment in the Application of Software Quality
Principles,"
MITRE Technical Report MTR-2874, Volume III,
The MITRE Corporation, Bedford, MA.
June 1974

Dahl O.J. et al.
"Simula 67 Common Base Language"
Technical Report, Norwegian Computing Center, Oslo
1968

Dahl O.J. and K. Nygaard
"Simula - An ALGOL Based Simulation Language"
Communications of the ACM, Vol. 9, No. 9
Sept. 1966

Dahl O.J., E. W. Dijkstra, C. A. R. Hoare,

Structured Programming,
Academic Press,

1972

Dijkstra E. W.,
"The Structure of the 'THE' Multiprogramming System,"
Communications of the AssociationComputing Machinery,
May 1968

Dudani S. and E.P. Stabler
"A Survey of Hardware Description Languages"
Syracuse University and Rome Air Developement Center
1978

Dunbar T. L.,
"JOCIT JOVIAL Compiler Implementation Tool,"
RADC-74-322,
January 1975

Elshoff J. L.,
"A Case Study of Experiences with Top Down Design and Structured
Programming,"
Research Labs, General Motors Corp., Warren, MI,
No. GMR-19742,
October 1974

Emery James C.
"The Coming Challenge of Campus Computation"
EDUCOM, Volume 20, No 3
pp. 20-24, Spring 1978

Feiler Peter H.
Incremental Program Construction
PhD thesis, Carnegie-Mellon University, Department of Computer Science,
1980

Felty James L., Martin S. Roth,
"High-Order Language Technology Evaluation: Software Support Tools,"
IR-204-2
Intermetrics Inc., Cabridge, MA,
October 1976

Flink Charles W. Flink, II
"A Microprogrammed Environment for a Software Development System"
Dahlgren Laboratory, Naval Surface Weapons Center
QM-1 Users Meeting Proceedings
March 1980.

Gordon Geoffrey
System Simulation
Prentice-Hall Inc., Englewood Cliffs, N.J.
1969

Graef N., Kretschmar, H., Loehr, K.-P., Morawetz, B.
"How to Design and Implement Small Time-sharing Systems Using Concurrent Pascal"
Software-Practice and Experience, Vol 9
pp. 17-24, 1979

Habermann Nico A.
"An Overview of the Gandalf Project"
CMU Computer Science Research Review 1978-79
1979

Heimbigner Dennis,
"Writing Device Drivers in Concurrent Pascal"
Operating Systems Review
Volume 12, No 4
October 1978

Howden W. E.,
"Experiments with a Symbolic Evaluation System,"
Proceedings of the National Computer Conference,
1976

Ivie E. L.
"The Programmer's Workbench -- A Machine for Software Development," Communications of the ACM, Vol. 20, No 10
p 746, Oct. 1977

Jensen E. D.,
"Private Communication"
Computer Science Department,
Carnegie-Mellon University
1980

Johnson S. C., M. E. Lesk
Language Development Tools
Bell System Technical Journal 57 (6)
July - August 1978

King J. C.,
"A New Approach to Program Testing,"
Proceedings of the International Conference on Reliable Software,
April 1975

Kiviat P.J.; R. Villauveva; H. Markowitz
The SIMSCRIPT II Programming Language
RAND Corporation, Santa Monica, R-460-PR
Oct. 1968

Klein Daniel
"MMPS--A Reconfigurable Multi-Microprocessor Simulator System,"
Proceedings of the National Computer Conference,
pp. 199-203, 1979

Lauer P. E.; Torrigiani, P.R.; Shields, M.W.
"COSY - A System Specification Language Based on Paths and Processes"
Acta Informatica 12,
pp. 109-158, 1979

Leverett Bruce W., et al
"An Overview of the Production Quality Coupiler-Compiler Project"
CMU-CS-79-105,
Carnegie-Mellon-University
1979

Lieblein Edward and Edith Martin
"Part V: Software for Embedded Computers"
Military Electronics/Countermeasures
pg 48, July 1979

Liskov Barbara "Primitives for Distributed Computing"
Lecture at Carnegie Mellon University
March 1980

London R.L.
"A View of Program Verification"
Proc. International Conference on Reliable Software
April 1975.

Mariani Michael P., and Palmer, David F.
"Tutorial: Distributed System Design"
IEEE Catalog No. EHO 151-1
Library of Congress No. 79-89494
1979

Mattsson Sven Erik
"Implementation of Concurrent Pascal on LSI-11"
Software - Practice and Experience, Vol 10
pp 205-217, 1980

McCauley E.J.; G.L. Barksdale; J. Holden
"Software Development Using a Development Support Machine (DSM)"
Proceedings of the Ada Environment Workshop, Harbor Island, San Diego, CA
November 27-29, 1979

McCauley E.J., and Drongowski, P.J.;
"KSOS -- The Design of a Secure Operating System"
Proceedings 1979 National Computer Conference, New York
p 345, 4-7 June 1979

McConnell T. et al.
"BUCS - I: A Software Reconfigurable Emulator System for DDP System Development and Algorithm Evaluation"
Proceedings of 1st International Conference of Distributed Computing Systems,
October 1979

McWilliams T. M.,
"The SCALD Timing Verifier: A New Approach to Timing Constraints in
Large Digital Systems"
Lawrence Livermore Laboratory,
University of Calif. and Computer Science Department,
Stanford University
1980

McWilliams T. M., L. C. Widdoes
SCALD "Structured Computer-Aided Logic Design,"
Proc. of the 15th Design Automation Conference,
Las Vegas, Nev.
June 1978

McWilliams T. M., L. C. Widdoes
"The SCALD Physical Design Subsystem,"
Proc. of the 15th Design Automation Conference,"

June 1978

Merilatt Randall L.; Leon J. Osterweil; Richard N. Taylor
"MUST Principles for ALICE"
Proceedings of the Ada Environment Workshop, Harbor Island, San Diego, CA
November 27-29, 1979

Miller E. F.,
"RXVP, An Automated Verification System for FORTRAN,"
Proceedings of Computer Science and Statistics:
8th Annual Symposium on the Interface,
February 1975

Mills H.
"Top-Down Programming in Large Systems"
Debugging Techniques in Large Systems
Prentice Hall Inc., NJ
1971

Newcommer J.M.; R.G.G. Cattell; P.N. Hilfinger; S.O. Hobbs; B.W. Leverett; A.H. Reiner; B.R. Schatz; W.A. Wulf
PQCC Implementor's Handbook
Internal Documentation, Carnegie-Mellon University, Computer Science Department
October 1979

Notkin David S. and Nico A. Habermann
SDS Documentation
1978

Osterweil Leon J.
"TOOLPACK: An Environment for Numerical Software Development"
Proceedings of the Ada Environment Workshop, Harbor Island, San Diego, CA
November 27-29, 1979

Parnas D. L.,
"On the Criteria to be Used in Decomposing Systems into Modules,"
Comunications of the Association of Computing Machinery,
December 1972

Parnas D. L.,
"A Technique for the Specification of Software Modules with Examples,"
Communications of the ACM,
May 1972

Powell M.S.
"Experience of Transporting and Using the SOLO Operating System"
Software - Practice and Experience, Vol. 9,
pp 561-569, 1979

Pretsker A. Alan,
The GASP IV Simulaion Language,
John Wiley,
New York,
1974

Ritchie D.M. and Ken Thompson
"The Unix Time Sharing System"
Communications of the ACM 17,
pp. 365-375, July 1974

Roubine O., and Robinson, L.
"SPECIAL Reference Maual, 3rd Edition",
Technical Report CSG-45,
SRI International, Menlo Park, CA
January 1977

Rochkind M. J.
"The Source Code Control System"
IEEE Transactions on Software Engineering
SE-1, 4
p 364, Dec 1975

Shiva Sajjan, G.
"Computer Hardware Description Languages- A Tutorial"
Proceedings of the IEEE, Vol 67, No. 12
Dec. 1979

Shrivastava S.K.
"Concurrent Pascal with Backward Error Recovery: Language Features and Examples"
Software - Practice and Experience, Vol 9
pp 1001-1020, 1979

Silberschatz A.; R.B. Kieburtz; and A.J. Berstein
"Extending Concurrent Pascal to Allow Dynamic Resource Management"
IEEE Transactions on Software Engineering, Se-3, No. 3
May 1977

SMITE installation and analysis manual,
TRW,
August 1977

Stevens W. P., G. J. Myers, and L. L. Constantine,
"Structured Design,"
IBM Systems Journal,
May 1974

Szewerenko Leland and William B. Dietz
NEBULA Instruction Set Architecture
Internal Documentation, Carnegie Mellon University,
Computer Science Department
March 25, 1980

Teichroew D., E. A. Hershey III, M. J. Bastarache,
"An Introduction to PSL/PSA,"
ISDOS Working Paper No. 86,
Univ. of Michigan, Ann Arbor, MI,
March 1974

Teledyne Brown Engineering
IORL, Users' Manual
1977

Tichy Walter F.
Software Development Control Base on Modular Interconnection
PhD theses, Carnegie-Mellon University, Department of Computer Science
1979

TRW-a "Phase II of Distributed Processing Architecture Design Program"
Technical Report
Contract # DASG60-79-C-0039,CDRL A004
Aug. 1979

TRW-b SEMANOL Reference Manual,
TRW Systems Group,
RADC Contract No. F30602-72-0047,
April 1973

Voydock Victor L.
"Software Engineering Facilities: Goals and a Practical Example"
Proceedings of the Ada Environment Workshop, Harbor Island, San Diego, CA
November 27-29, 1979

R. E.; B. A. Clausen II
"Software First: Our Viking Experience"
Summer Computer Simulation Conference
Newport Beach, Calif., July 1978.

Weinberg G. M.,
The Psychology of Computer Programming,
New York, Van Nostrand Reinhold,
1971

Wirth N.,
"Program Development by Stepwise Refinement,"
Communications of the Association of Computing Machinery,
April 1971

Wirth N.,

Wittie 79 Larry D.
"A Distributed Operating System for a Reconfigurable Network Computer"
Proceedings of the IEEE,
June 1979